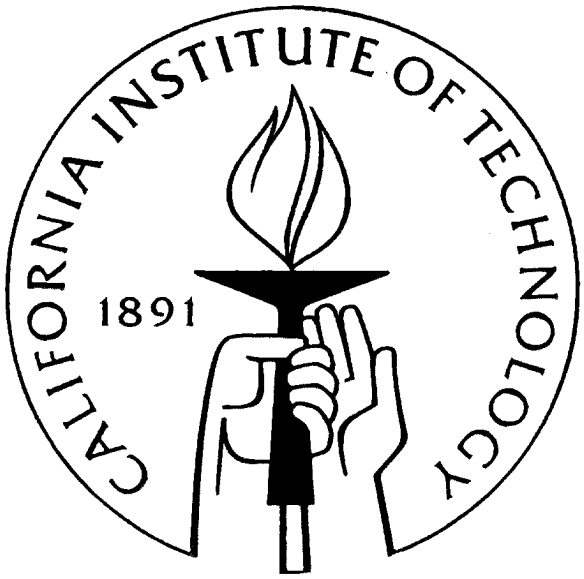

On Detection and Generation of Deadlock-free Reshuffling in the VLSI Synthesis Method

Jiazhao Jessie Xu

Computer Science Department
California Institute of Technology

Caltech-CS-TR-96-10

On Detection and Generation of Deadlock-free Reshuffling in the VLSI Synthesis Method

Thesis by
Jiazhao Jessie Xu

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

May 26, 1995
revised on July 31, 1995

Contents

List of Figures

Chapter 1

Introduction

1.1 Why Study Reshuffling

With the quick growth in the scale of VLSI circuits and the accordingly increasing complexity in the design of those large systems, the correctness of the design has demanded designers' attention. There are (recent) examples of microprocessors which have "bugs" in the designs and this raises questions about their reliability in performance. Personally, I am strongly against the popular use of word "bug" in the computer science world when people refer to their *mistakes* in the design. *Mistake* as defined by Webster is a wrong action or statement proceeding from faulty judgment, inadequate knowledge, or inattention; so it's not like a bug which crawls into our system on its own and ruins our life, but is a reflection of wrong decisions or of the inadequacy in our design methodology.

This treatise is about research work on *reshuffling*, a specific transformation used in Dr. Alain Martin's method of designing asynchronous circuits [8]. I try to address the effect of reshuffling on one of the important correctness properties of a circuit – the absence of deadlock.

My interest in this theoretical area of research stems from the conviction that correctness is one of the most important aspects in the VLSI design. Correctness should be taken care of at the design phase instead of the testing phase; furthermore I don't believe there exists some general simple and complete test method to prove the correctness of a complicated VLSI system. Therefore I am attracted by Dr. Martin's idea of preserving the correctness through the derivation of the physical layout of a circuit from a high level

specification, i.e. each transformation step perserves the correctness properties.

Second, my interest in this topic stems from the fact that reshuffling is the only transformation step in Martin's method for which correctness properties will not always be preserved; a bad reshuffling could result in deadlock while a good one improves the system's performance, especially efficiency, greatly. The performance improvement that reshuffling brings to the design makes it an indispensible transformation in the synthesis method that is used extensively in almost all designs. In order to be able to benefit from this transformation without putting the correctness at risk, we need to study how to reshuffle in a safe way by either defining this transformation more restrictively so that the unsafe ones are excluded, or by checking certain properties which guarantee that correctness is preserved by transformation, or a combination of both.

Finally, absence of deadlock (or "liveness" of the system) is one of the most important correctness requirements of a system. Liveness addresses the progress aspect of the system. It is tightly bound with the global system behavior which makes the analysis very complicated due to the inevitable need to model the system behavior. The environment description/modeling adds greatly to the complexity because the set of system behaviors is decided by the set of allowable environment actions. With the synchronous system, the system modeling can be done through timed Petri-nets theoretically; however, it is hard to define deadlock when the time is involved. With the asynchronous system, especially that designed using Dr. Martin's transformation methodology, the system behavior is essentially boiled down to the sequencing of transition signals; so the absence of deadlock could be viewed as a property satisfied by the sequencing relationship – a property of logic, and the analysis can be performed on the handshake expansion description level.

1.2 Dr. Martin's Method

In the late 1980's, Dr. Alain Martin developed a beautiful and revolutionary methodology for the design of *delay-insensitive circuits*, a subclass of asynchronous circuits where propagation delays in wires and gates are assumed to be arbitrary and unbounded (but finite) [8]. The beauty of the synthesis

method lies in the series of semantics-preserving transformations (except the reshuffling phase) which derives the delay-insensitive circuit, the physical realization, from the top-level specification.

1.2.1 CSP Specification and Its Refinement

The synthesis method starts from a simple top-level specification using C.A.R. Hoare's CSP language [5] (CSP stands for *Communicating Sequential Processes*), which involves very few communicating processes so that the correctness is easily analysed. This top-level CSP is further decomposed to a refined version of the CSP program after making major design decisions such as pipelining which crucially influence a chip's efficiency. The decomposition follows a set of rules which preserve the semantics and thus the correctness as well.

1.2.2 HSE and Reshuffling

The refined CSP program is then transformed into another description, the *handshaking expansion*, using a certain communication protocol (four phase handshaking protocol is used for most communication actions while occasionally two phase handshaking protocol is used as well). The final program is a composition of transition signals, composed sequentially, in parallel, and by selection. This phase of transformation hardly involves any major decisions, and I call it the *directly-derived handshaking expansion* (or directly-mapped handshaking expansion).

Unfortunately the directly-derived HSE produces inefficient circuits in almost all designs, therefore reshuffling is performed to transform the directly-derived HSE into a more efficient HSE. However this transformation, reshuffling, changes the semantics and there are hardly any explicit rules to describe how to reshuffle correctly and with improvements in efficiency, nor simple ways to check the correctness or to evaluate the efficiency. Realizing the nonnegligible importance of reshuffling for the correctness and performance of the circuit, I studied the influence of reshuffling on correctness, and that is what this treatise is all about.

1.2.3 PRS and the Circuit Realization

After obtaining the reshuffled HSE, the next transformation produces the *production rule set*, which is the canonical representation of a digital circuit. This transformation is formalized and correctness can be checked.

Then the production rule representation can be decomposed into equivalent networks of digital operators, depending on the set of building blocks used and the technology applied (which is CMOS here, though it works for other technologies such as Gallium Arsenide).

1.3 Topics in this Treatise

The following is a chart of the transformation steps in Dr. Martin's synthesis method:

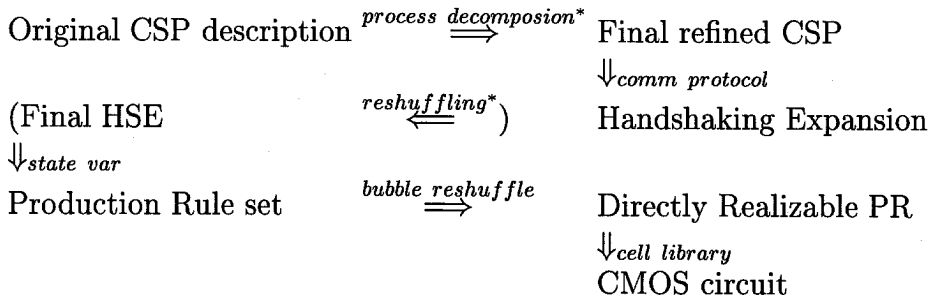

Figure 1.1: Transformation Relations in the Martin Synthesis Method

The transformation stage studied in this treatise is reshuffling on the handshaking expansion level. A *safe reshuffling*, informally speaking, is a reshuffling that transforms a handshaking expansion into another semantically-equivalent handshaking expansion. As a liveness property of the system, absence of deadlock should be preserved through a safe reshuffling. In most cases, unsafe reshuffling results in deadlock for the resulting system. In this treatise, we focus on the following questions :

- How to check if a reshuffling is safe or not.
- How to produce safe reshuffling.

1.4 Outline of the Treatise

During my preliminary research, the most important lesson I learned is that an unclear and improper specification is the poison of all work. As an outsider who entered the computer science area three years ago, I felt a lot of frustration caused by the careless usage of terminology: multiple meanings for the same name, and ill-defined terms. After realizing the impraticality of straightening the system and the vitality of using precise terms for theoretical proofs, I decided that the best thing for me to do is to define terms in the way that I will use them in this treatise.

Part 0 (Chapter 1): Introduction

Part I (Chapter 2 – Chapter 5): specifications, definitions and important concepts related to reshuffling and deadlock. These contribute to the aforementioned purpose of building a solid base for the application of the formal methods.

Part II (Chapter 6 – Chapter 8): theories for checking and generating deadlock-free reshuffling.

1.5 Note on Notation

In the thesis, I refer to Dr. Alain Martin's synthesis method as *the synthesis method* (or the Martin synthesis method).
The circuits by default are delay-insensitive circuits with the isochronic fork assumption (these are sometimes called *quasi-delay- insensitive circuits*), the technology used is the CMOS technology.
Some abbreviations:

- CSP – Communicating Sequential Processes, a language by C.A.R. Hoare
- HSE – Handshaking Expansion
- PR – Production Rule

For assignment we use ":=", i.e. $x := 0$.
For equality we use "=", i.e. $x = 1$, which means the present value of x is 1. Sometimes this could be confused with the comparison operator in a boolean function, therefore judgment should be made according to the context. In a

guarded command, $x = 1$ always represents a boolean funtion. One explicit way to avoid this confusion is to write each boolean comparison as $\{x = 1\}$.

Chapter 2

Handshaking Expansion

2.1 What Is Handshaking Expansion

Handshaking expansion is an intermediate level representation of the circuit in the synthesis method. It can be derived from the CSP specification by using some heuristics, for example the communication protocol, data encoding etc.

2.1.1 Example

Consider the CSP for the control of a one-place buffer:

$$*[L; R]$$

Once we have decided that channel L uses the passive four-phase handshaking protocol and that channel R uses the active four-phase handshaking protocol, i.e. L is implemented as $[li]; lo\uparrow; [\neg li]; lo\downarrow$ and R as $ro\uparrow; [ri]; ro\downarrow; [\neg ri]$, then it is a straightforward matter to transform the CSP for the buffer into the following handshaking expansion:

$$*[[li]; lo\uparrow; [\neg li]; lo\downarrow; ro\uparrow; [ri]; ro\downarrow; [\neg ri]]$$

Another widely used heuristic is the dual-rail data encoding [10].
Consider the following assignment, where x is a local register and y is transmitted data:

$$x := y$$

The dual-rail encoding of y gives the following HSE:

$$y0 \longrightarrow x\downarrow \parallel y1 \longrightarrow x\uparrow$$

The transformation from the CSP to the handshaking expansion depends greatly on the heuristics, especially on the interfaces. For example, the CSP for a one-place buffer with data transfer is:

$$*[L?x; R!x]$$

the resulting handshaking expansion would be entirely different depending on whether the data and control are separate or not, whether the ports are passive or active etc.

2.2 Constructs of Handshaking Expansion

2.2.1 Terms in Handshaking Expansion

The *terms* (or basic elements) in a handshaking expansion are transitions and waits defined as follows:

Def: A *transition* is an assignment to a variable to change its value. An *up-going transition* on variable x means to assign $x :=$ **true**, denoted also as $x\uparrow$. A *down-going transition* on variable x means to assign $x :=$ *false*, denoted as $x\downarrow$. We call **skip** the *null transition.*

Note: A transition is an event of non-vacuous firing, i.e. a transition will change the value of the variable. It is actually a triplet $\langle x, \updownarrow, n\rangle$ where x is the variable, $\updownarrow$ is either the up-going transition $\uparrow$ or the down-going transition $\downarrow$, and n is an integer which represents the number of occurrences of the *type of transition* $\langle x, \updownarrow \rangle$.
(end of Note)

Def: A *wait* $[G]$ is a wait for the boolean expression G to become **true**. In particular when G is a variable x, $[x]$ means to wait till $x =$ **true**; while $[\neg x]$ means to wait till $x =$ **false**. We call [**true**] the *null wait.*

2.2.2 Composition Rules for Statements

A *statement* is the sequential and parallel composition of terms and statements. Terms can be regarded as the simplest form of statements. Therefore we can say statements are closed under sequential and parallel composition.

Def: *Sequential composition* of two statements S_1, S_2 is a binary operator ";" such that the composition $S_1; S_2$ yields another statement which means

S_2 won't start till the completion of S_1. Sequential composition is associative but not commutative.

Def: *Parallel composition* of two statements S_1, S_2 is a binary operator "$\|$" such that the composition $S_1 \| S_2$ yields another statement which means the concurrent execution of statements S_1 and S_2. Parallel composition is weakly fair. It is associative and commutative.

2.2.3 Guarded Commands

The concept of a guarded command has been proposed by Dr. Edsger W. Dijkstra in [2].

Def: A *guarded command* $G \longrightarrow C$ consists of the guard G, a boolean expression, and a program part C, which could be a statement or a construct of the guarded command. It means that C can be executed only when G assumes the value **true**. The semantics of the guarded command is combined with the control constructs: selection and repetition, which we will define later.

Note: One mistake easily made here is to assume that if G is true then C will be executed. The execution of the guarded command $G \longrightarrow C$ depends on the control constructs for the guarded command. Therefore a guarded command *is not* a program part because it has no operational meaning of its own.
(end of Note)

2.2.4 Control Constructs

We will give the operational definitions of the following control constructs:

Def: *Deterministic selection* is a control construct on n guarded commands

$$[G_1 \longrightarrow C_1 \| \ldots \| G_n \longrightarrow C_n]$$

which operates as follows:
At any time at most one guard holds (has the value **true**), and the guarded command of the true guard will be executed; if none of the guards is **true** then the execution of the whole selection command will be suspended until one of the guards becomes **true**.

A statement S can be regarded as the special deterministic selection: [**true** $\longrightarrow S$]. A wait $[G]$ is another special case: $[G \longrightarrow$ **skip**]. Any deterministic selection of the form $[G \longrightarrow C]$ is equivalent to "$[G]; C$" if G remains **true** when the execution of C starts.

A slightly different construct is the non-deterministic selection command:

Def: A *non-deterministic selection* is a control construct on n guarded commands

$$[G_1 \longrightarrow C_1 | \ldots | G_n \longrightarrow C_n]$$

which operates as follows:
At any time more than one guard could hold, then an arbitrary true guard is selected for execution of the corresponding guarded command; if none of the guards is **true** then the execution of the whole selection command will be suspended until one of the guards becomes **true**.

Another very important control construct frequently used is that of repetition:

Def: A *deterministic repetition* command

$$*[G_1 \longrightarrow C_1 \| \ldots \| G_n \longrightarrow C_n]$$

is executed as follows:
At any time, at most one guard holds. The program will repeatedly select the true guard and execute the corresponding guarded command until none of the guards is **true**.

Similar to the non-deterministic selection command, we have the non-deterministic repetition command

$$*[G_1 \longrightarrow C_1 \| \ldots \| G_n \longrightarrow C_n]$$

where at any time there could be several true guards and an arbitrary selection is required.

A special case of the repetition construct is $*[C]$, where C is a program part. It can be written in the more recognizable format as: $*[$**true** $\longrightarrow C]$.

2.3 Processes

The word "process" is one typical example of abused terminology. In software, people refer to a process as a batch of work. In VLSI design, the

program to describe the entire chip, i.e. the microprocessor, can be called a process; while sometimes from a microscopic point of view, the microprocessor consists of many processes executing concurrently, i.e. the memory unit, the fetch unit, the exec unit and ALU etc.

2.3.1 A Definition

Since our goal here is to study sequential processes which execute concurrently but communicate with each other through message passing, I define the meaning of process in this thesis as follows:

Def: A *process* is a program part that is not the parallel composition of two smaller program parts.

For example, $*[S_1] \parallel *[S_2]$ is not a process but a composition of two processes: $*[S_1]$ and $*[S_2]$.

This definition is well-defined because parallel composition is associative and commutative.

2.3.2 Variable Sharing

In strict communicating process design style, a variable is local to a process, noncommunicational variables are not shared between processes. However, in some designs, including the design of the first Caltech Asynchronous Microprocessor, this taboo is violated in a trade for simplicity and performance, and a variable of one process may be inspected by another process.

This violation of the locality rule doesn't bring us much trouble as long as the sharing is restricted to only multiple reading of the shared variable. Essentially, communication variables can be regarded as shared variables as well. The communication protocol guarantees the correctness of the sharing. Therefore, variable sharing could be taken care of properly; for simplicity, I assume that there are no shared variables between processes in this thesis.

2.4 Straight-line Handshaking Expansion

Def: A *straight-line handshaking expansion* is a process of the form $*[S]$, where S is a statement constructed solely from sequential composition of

terms.

For example, we have the two-to-four phase converter:

$$converter \equiv *[[li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\downarrow]$$

Sometimes two consecutive waits are combined into a single wait. For example, $[\neg li]; [ri]$ can be combined into $[\neg li \wedge ri]$. This alteration won't affect the system behavior, because the waits don't change the state of the machine, and we only care about the program state after the waits are completed and before the subsequent transition is fired.

Chapter 3

Reshuffling

3.1 Communication

Communication is an important method used among processes to pass (data) messages or to achieve synchronization. All (sequential) processes are connected with each other via channels. Each channel has ports to other processes; in this treatise I assume that every channel has exactly two ports connected to two processes. Processes communicate with each other by using the communication commands on the ports. Communication actions on the two ports of the same channel have to be synchronized. Processes execute their respective local programs until communications are initialized; a synchronization is a pair of communications (X, Y) that says the completion of X coincides with the completion of Y. Details of the semantics of synchronization are studied in [7].

Def: For each channel between two processes, there is a pair of communication (X, Y) being performed on the channel (sometimes, we denote X also as port, and (X, Y) as channel connecting port X and Y). A *communication action* on X refers to an occurrence of communication X.

3.1.1 Four-phase and Two-phase communication

Channel (X, Y) is implemented by two wires (xo $\underline{w}$ yi) and (yo $\underline{w}$ xi),and xo, xi, yo, yi are called the *handshaking variables of* (X, Y). A transition on handshaking variables (or communication variables) is a *communication transition*, i.e. $xo\uparrow$.

Let

$$U_x \equiv xo\uparrow;[xi] \qquad D_x \equiv xo\downarrow;[\neg xi]$$
$$U_y \equiv yo\uparrow;[yi] \qquad D_y \equiv yo\downarrow;[\neg yi]$$

Def: A *two-phase handshaking protocol* is a communication protocol for a pair of communications (X, Y) that implements the communication X as a sequence of alternating U_x and D_x, and implements Y as a sequence of alternating U_y and D_y.

Def: A *four-phase handshaking protocol* is a communication protocol for a pair of communications (X, Y) that implements X as $U_x; D_x$ and Y as $U_y; D_y$.

3.1.2 Four-phase Handshaking Protocols

Notice that the implementation for X and Y aren't symmetrical either in the two-phase and the four-phase protocol. The way that X is implemented is called *active* and the way Y is implemented is called *passive.* The two-phase communication is more efficient than the four-phase communication where the D_x/D_y part is used just for resetting. But the two-phase communication can't be applied in most cases because of the impossibility of syntactically determining which X or Y actions follow each other in an execution. Therefore,the four-phase handshaking protocol is adopted in most designs for the purposes of correctness and simplicity. When the operator delays dominate the communication costs, four-phase handshaking usually leads to the more efficient solutions. Great efficiency is achieved when reshuffling is applied to rearrange the resetting part of the four-phase handshaking protocol.

There are three types of four-phase handshaking protocol:

- **Active four-phase protocol**:

$$X \equiv xo\uparrow;[xi];xo\downarrow;[\neg xi]$$

- **Passive four-phase protocol**:

$$X \equiv [xi];xo\uparrow;[\neg xi];xo\downarrow$$

- **Lazy-active four-phase protocol**:

$$X \equiv [\neg xi];xo\uparrow;[xi];xo\downarrow$$

3.2 Definition of Reshuffling

3.2.1 Historical Study and Results about Reshuffling

The introduction of reshuffling into the synthesis methods can be traced back to the following description [8].

$$X = U_x; D_x \qquad Y = U_x; D_x$$

Property 2 In X and Y, D_x and D_y are used only to reset all variables to **false**. Hence, provided that the cyclic order of the actions of X and Y is maintained, the sequences D_x and D_y can be inserted at any place in the program of each of the processes without invalidating the semantics of the communication involved. This transformation, called reshuffling, may introduce a deadlock.

Reshuffling, which is the source of significant optimization, will be used extensively. It is therefore important to know when Property 2 can be applied without introducing deadlock.

Later on, another major advantage of reshuffling was discovered: reshuffling sometimes can turn an unsafe handshaking expansion, i.e. an HSE with indistinguishable states in it, into a safe handshaking expansion where each state is unique.

It has always been an interesting and important question how to pick and generate those reshufflings that will not introduce any deadlock; I refer to these as *deadlock-free reshufflings* for a given handshaking expansion. Two simple cases of deadlock-free reshufflings were presented in [11].

There are two simple cases where the reshuffling of sequence "$U_x; D_x; S$" into sequence "$U_x; S; D_x$" does not introduce deadlock:

- S contains no communication action, or
- X is an internal channel introduced by process decomposition.

During the early stage of this research on deadlock-free reshufflings, I expected to find more "simple cases" as cited above where DF-reshufflings depend on little system information; unfortunately this simple class of DF-reshufflings is very limited. I will discuss more about this class of DF-reshufflings in Chapter 8.

3.2.2 Examples of Reshuffling

Let's study the example of a commonly used process introduced by process decomposition, the *call* operator, denoted as (L/R). The CSP for (L/R) is $*[[\bar{L} \longrightarrow R; L]]$. L is an internal channel introduced for process decomposition and assumes the passive four-phase handshaking communication protocol; R takes the active four-phase handshaking communication protocol. We have the following directly derived handshaking expansion for (L/R):

$$*[[li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; lo\downarrow]$$

This HSE is unsafe because we have the presence of indistinguishable states: the state right before $ro\uparrow$ and the state right after $ro\downarrow$. State variables are needed to turn this HSE into a safe one.

Def: A *safe handshaking expansion* is an HSE that can be transformed into a correct production rule set that correctly models the handshaking expansion (the set of program variables in the HSE and PRS are the same).

One reshuffling we can perform here is to pospone the second half of the four-phase communication R:

$$*[[li]; ro\uparrow; [ri]; lo\uparrow; [\neg li]; ro\downarrow; [\neg ri]; lo\downarrow]$$

The above reshuffling results in a safe reshuffling where all the states are distinguishable, and the final implementation of the process reduces to only two wires.

3.3 Correctness of Reshuffling

Assume the original CSP program is correctly specified, then the process decomposition and further transformation to the handshaking expansion using the four-phase handshaking is correct too, this renders a correct directly-derived HSE program of the system. A reshuffling of the directly-derived HSE program is correct if all valid environments for the original system can not observe any difference after replacing the original HSE with the reshuffled HSE [4]. This requires that the reshuffling not introduce any deadlock and not change the data-dependency relations for all possible computations.

In this treatise, I will study the most important aspect of a correct reshuffling – absence of deadlock.

Chapter 4

Partial Orders

In this Chapter, I will establish the mathematical foundation for future work: concepts and properties of partial order relations. In asynchronous VLSI design, the successor relation is a critical concept that decides the nature of the circuit – whether it is strictly delay-insensitive or not. First I will show that the successor relation is a partial order relation on transitions, second, that it also facilitates the analysis of liveness.

4.1 Definition of a Partial Order

A *partial order* on a set S is a binary relation $\leq$ defined on a subset of $S \times S$ that is reflexive, transitive and antisymmetric. If $\leq$ is a partial order on S we say that the pair $(S, \leq)$ is a *partially ordered set*. When there is no ambiguity we simply say that S is a partially ordered set. If x and y are elements of the partially ordered set S, we write $x < y$ when $x \leq y$ but $x \neq y$ ([12]).

Explicitly, a partially ordered set $(S, \leq)$ is a set of partial order relations closed under reflexivity, transitivity, and antisymmetricity.

4.2 Successor Relationship

The successor relation was first proposed by Dr. Alain Martin in his beautifully-written paper [9]: The Limitation to Delay-Insensitivity in Asynchronous Circuits. The successor relationship was defined on the production rule set. Here I will give my definition of the successor relation with small modifications from the original one for clarity and formality.

4.2.1 Definition

Def: A *guard* (in a production rule) in this thesis is a boolean expression in disjunctive normal form. Each disjunct is a conjunction of literals, each *literal* is either the variable itself, i.e. x, or the negation of the variable, i.e. $\neg x$.

Note: A literal is the simplest boolean expression of a boolean variable x which takes the form either x (sometimes refered to as positive form of x) or $\neg x$ (negative form of x). Don't confuse the literal x with $\{x\}$ (i.e. $\{x = \textbf{true}\}$) or $\neg x$ with $\{\neg x\}$ (i.e. $\{x = \textbf{false}\}$).
(end of Note)

Def: In the guard of a production rule, i.e. $G \longrightarrow x\uparrow$, when a literal is evaluated to be **true** at a certain program stage there must be a transition on the variable associated with the literal which performed the assignment. We call this transition a *cause* on the literal evaluated to be **true**, for example: a transition of type $a\uparrow$ is the cause on the literal a when $\{a(= \textbf{true})\}$ holds, and a transition of type $a\downarrow$ is the cause on the literal $\neg a$ when $\{a = \textbf{false}\}$ holds.
When the guard of a production rule is evaluated to be **true** and causes a non-vacuous firing (a transition) on the target variable, we attach a set of *causal transitions* to the set of literals with value **true**. The transition on the target variable is a *successor* of each of the causal transitions.

Note: In Dr. Martin's paper, all transitions are not indexed, whereas in this paper such unindexed notation indicates the type of transition.
(end of Note)

4.2.2 An example

Suppose we have the production rule:

$$(a \wedge b) \vee (c \wedge d) \longrightarrow x\uparrow$$

When a transition (a non-vacuous firing) on x happens due to the literals a and b being **true**, then the transition $\langle x\uparrow, i\rangle$ is a successor of transition $\langle a\uparrow, j\rangle$ and of transition $\langle b\uparrow, k\rangle$. In another compution, transition on x happens when the literals c and d are true, then the transition on x is a successor of the transitions on c and d.

4.2.3 Unique Successor Set Criterion

The USS (Unique Successor Set) criterion is very important in deciding whether a given circuit, a network of gates, is strictly delay-insenstive or not. It is a beautiful theorem with appealing simplicity (details see [9]).

Def: In a computation, the *successor set* of a transition t is the set of variables for which a transition on each variable is a successor of t.

Def: A *computation has the unique-successor-set (USS) property* when all non-final transitions on the same variable have the same successor set. A set of computations has the USS property when all non-final transitions on the same variable have the same successor set in all computations of the set.

Theorem 4.2.1 (USS Theorem) *A set of computations of a DI circuit has the USS property.*

With the USS property, we can lift the sucessor relationship from the transition set to type of transition set (i.e. with index omitted), further more to the variable set (i.e. with the transition type omitted).

4.3 Successor Relation and Partial Order

Let's define $\langle x\updownarrow, i\rangle \prec \langle y\updownarrow, j\rangle$ to mean that there exists a finite chain of transitions $t_i,\ i = 1, \ldots, n$ with $t_1 = \langle x\updownarrow, i\rangle$ and $t_n = \langle y\updownarrow, j\rangle$ and t_{i+1} a successor of t_i. The successor relation is a pre-order relation $\prec$ where transitivity and anti-reflexivity holds (i.e. $\langle x\updownarrow i\rangle \prec \langle x\updownarrow, i\rangle$ is always false).

Now let's refine (= re-define) the successor relation so that it forms a partial order: $\langle x\updownarrow, i\rangle \preceq \langle y\updownarrow, j\rangle$ means $\langle x\updownarrow, i\rangle \prec \langle y\updownarrow, j\rangle$ or $\langle x\updownarrow, i\rangle = \langle y\updownarrow, j\rangle$ (i.e. $x\updownarrow = y\updownarrow, i = j$).
The successor relation satisfies reflexivity:

$$\langle x\updownarrow, i\rangle \preceq \langle x\updownarrow, i\rangle$$

The successor relation satisfies transitivity:

$$(\langle x\updownarrow, i\rangle \preceq \langle y\updownarrow, j\rangle) \wedge (\langle y\updownarrow, j\rangle \preceq \langle z\updownarrow, k\rangle) \Rightarrow \langle x\updownarrow, i\rangle \preceq \langle z\updownarrow, k\rangle$$

The successor relation satisfies antisymmetry:

$$(\langle x\updownarrow, i\rangle \preceq \langle y\updownarrow, j\rangle) \wedge (\langle y\updownarrow, j\rangle \preceq \langle x\updownarrow, i\rangle) \Rightarrow \langle x\updownarrow, i\rangle = \langle y\updownarrow, j\rangle$$

Note: The successor relation is different from the successor. For example, given transitions t and s with a successor relation $t \prec s$, it is not necessary that t be a successor of s.
(end of Note)

4.4 Generators of a Partially Ordered Set

Because of the transitivity of partial orders, we can generate new partial order relations (abbreviated as "partial orders") from exisiting ones. Therefore we can represent a partially ordered set by a subset of partial orders which can generate the entire partial order in the partially ordered set, and we want this subset to be as small as possible. First, let's define some useful concepts related to partial orders:

Def: A *complete set of partial orders* refers to a set of partial orders that is transitive-closed, i.e. any partial order generated by using a subset of partial orders is included in the set.

A partially ordered set is a complete set of partial orders.

Note: In this thesis, I will try not to refer to a set of partial orders as a *partial order set*, to avoid confusion with partially ordered set. In case, I use the term partial order set I mean a set of partial orders.
(end of Note)

Def: An *irreducible set of partial orders* is a set of partial orders such that no singleton in the set can be generated from any disjoint subset of partial orders.

For example: $\{\langle x\updownarrow, i\rangle \preceq \langle y\updownarrow, j\rangle, \langle y\updownarrow, j\rangle \preceq \langle x\updownarrow, i\rangle, \langle x\updownarrow, i\rangle = \langle y\updownarrow, j\rangle\}$ is a complete set of partial orders, $\{\langle x\updownarrow, i\rangle \preceq \langle y\updownarrow, j\rangle, \langle y\updownarrow, j\rangle \preceq \langle x\updownarrow, i\rangle\}$ is an irreducible set of partial orders, $\{\langle x\updownarrow, i\rangle = \langle y\updownarrow, j\rangle\}$ is another irreducible set of partial orders.

Def: A *generating set of a partially ordered set* is a maximum irreducible subset of partial orders.

Also it is easy to show that a generating set of a partially ordered set is a minimal subset that generates the whole partially ordered set.

In the above example, $\{\langle x\updownarrow, i\rangle \preceq \langle y\updownarrow, j\rangle\}$ is an irreducible subset of

partial orders but *not* a generating set of the partially ordered set. But $\{\langle x\updownarrow, i\rangle \preceq \langle y\updownarrow, j\rangle, \langle y\updownarrow, j\rangle \preceq \langle x\updownarrow, i\rangle\}$ is a generating set, so is $\{\langle x\updownarrow, i\rangle = \langle y\updownarrow, j\rangle\}$.

Def: Given a partially ordered set $(S, \leq)$, we can decompose the set S into disjoint subsets $S_1, \ldots, S_n$ with respect to the partial order $\leq$ such that

1. for any element x in a subset S_i, there exists at least one other element y in the subset S_i such that either $x \preceq y$ or $y \preceq x$ holds
2. for any element x in a subset S_i, there exists no partial relation between x and y if y belongs to some subset S_j for $j \neq i$.

$S_1, \ldots, S_n$ is called a *decomposition* of S with respect to partial order $\leq$, it can be written as $(S, \leq) = \biguplus_{i=1,\ldots,n}(S_i, \leq)$. And subsets $(S_i, \leq), (S_j, \leq)$ (where $i \neq j$) are said to be *independent* of each other.
(end of definition)

For a finite partially ordered set $(S, \leq)$, there exists a finest decomposition of $(S, \leq)$. Furthermore, any decomposition of $(S, \leq)$ can be eventually decomposed to this finest decompostion. Therefore the finest decomposition of a finite partially ordered set is unique.

Theorem 4.4.1 *Let $(S_1, \leq), \ldots, (S_n, \leq)$ be a decomposition of the partially order set $(S, \leq)$, then a generating set of $(S, \leq)$ is the (disjoint) union of generating sets of subsets $(S_i, \leq)$, $i = 1, \ldots, n$.*

Proof: First let's show the union of generating sets of the $(S_i, \leq)$ forms a generating set of the partially ordered set $(S, \leq)$. It is trivial that the union of generating sets of the $(S_i, \leq)$ generates the entire partially ordered set $(S, \leq)$. And it is also irreducible: suppose it can be reduced, then one of the generating sets of some $(S_k, \leq)$ can be reduced which contradicts the definition of generating set.

Next let's show that any generating set G of $(S, \leq)$ can be written as the union of $G_1, \ldots, G_n$ where G_i is a generating set of $(S_i, \leq)$. Let's decompose G into n parts where G_i belongs to $(S_i, \leq)$, and $G = \biguplus_{i=1,n} G_i$. We claim that G_i is a generating set of $(S_i, \leq)$. For each G_i, G_i is irreducible so there exists a generating set G_i' of $(S_i, \leq)$ with G_i as a subset. Therefore $\biguplus_{i=1,n} G_i'$ (where $G_i' \supseteq G_i$) constitutes a generating set of $(S, \leq)$ from the first part we just proved. Then we must have $\biguplus_{i=1,n} G_i = \biguplus_{i=1,n} G_i'$, and therefore $G_i = G_i'$. $\square$

4.5 Derive Successor Relation from HSE

Since the handshaking expansion gives complete information on the sequencing of transitions on variables given a fixed environment scenario, theoretically we could generate the successor relations for any computation from a handshaking expansion. I will present an algorithm for this generation in Chapter 6. This saves us the transformation from the HSE to PRS to obtain the successor relation. And furthermore, the algorithm generates the generating set for the partially order set of transitions.

Chapter 5

Computation, Environment and Deadlock

5.1 State of a Program

Def: A *state of a program* is a function from the set of program variables to booleans. It can be written as a vector of assertions on all program variables with a subscript identifier, i.e. $\langle\{x\},\ldots,\{\neg y\}\rangle_\sigma$, or $\sigma(x,\ldots,y) = \langle 1\ldots 0\rangle$. Sometimes the ordered set of program variables is fixed and so a state can be abbreviated as a boolean vector with a subscript identifier, i.e. $\langle 1\ldots 0\rangle_\sigma$. The dimension of the boolean vector equals the number of program variables. The *state of a variable* is the boolean value of the variable at a program state, i.e. $\sigma(x) =$ **true** means that the value of x at the program state σ is **true**.

For example, in the buffer case, the set of program variables are li, lo, ri, ro. In the initial state, all the variables take the value **false**, therefore the initial state is $\langle 0000\rangle_{init}$.

5.2 State Transition

A state can be (and only be) changed to another state by transitions on program variables. For a handshaking expansion, some state changes are due to transitions on local variables or output variables of the HSE, and some are due to transitions on the input variables. For the transitions on the input variables, the moment that the HSE observes the state change could occur after an indefinite time lapse from the moment that the transition on the in-

put variable has ocurred, which could cause an undefined state on the input variables at certain moments.

Def: The *fundamental mode* of execution refers to those executions where each state change only involves one transition on a program variable, i.e. no two transitions can happen simultaneously.

Because the circuits studied in this thesis are delay-insensitive, the fundamental mode assumption actually applies to possible circuit executions. Using this assumption, we could build the state transition graph for a computation. A state transition graph is a directed graph whose nodes are states and whose edges are transitions (detail see [6]).

5.3 Defintion of Computation

In [9], computation is defined by Dr. Martin as follows:

> A *computation* is a particular successor relation on a set of transitions, such that each computation corresponds to a possible execution of the circuit.

A computation is different from an execution.

Def: An *execution* is a trace of states under the fundamental mode assumption.

A computation, however, could correspond to a set of executions (or a set of traces). Interleaving won't change a computation; a computation can only be changed by initial states and environments.

5.4 Environment

For each system P, we must specify a set of valid environments $\{E_1, E_2, \ldots, E_n\}$ in which the system will behave as specified. I call the E_i *environment scenarios.* Therefore, given an initial state of the system and a deterministic evironment scenario, we have a unique computation for the system. In this treatise, I will restrict the environment scenario to deterministic ones.

Def: Given an (open system) P, and a valid environment scenario of P: E, $P \parallel E$ forms a *closed system.*

For example, if we have the CSP program for a one-place buffer: $*[L; R]$, then we can have the following environment scenarios: $*[L'] \parallel *[R']$ or $*[L'; R']$, and the corresponding closed systems: $*[L; R] \parallel (*[L'] \parallel *[R'])$ or $*[L; R] \parallel *[L'; R']$.

5.5 Progress and Deadlock

Progress is an important correctness property of a program. The progress of a correctly implemented system requires:

- Given a finite computation, the program will eventually end up at some terminating state.
- Given an infinite computation, the program will eventually change its state.

Notice the second progress requirement doesn't capture the livelock situation because in a *livelock* situation there exists a cycle of state transitions. However, both of the two requirements capture the deadlock situation.

Though deadlock is a major violation of the progress property, we can't define deadlock for an asynchronous system by just taking the negation of the progress conditions because we can't tell when the whole system stalls at some state due to delay or due to a deadlock situation.

Def: *Deadlock* is a non-terminating state $\perp$ in a program execution at which no transition can be executed to change the state of the program.

Marcel Van de Goot gives a more formal definition of deadlock in his Ph.D. thesis by using execution trees[4].

Chapter 6

Deadlock Detection on Systems with SLHE Processes

The march of a thousand miles begins with the first step.
– ZeDong Mao, Chairman Mao's Sayings and Thoughts

6.1 Introduction

In this Chapter, I will study the detection of deadlock-free reshuffling of a closed system where all the processes are modeled by the straight-line handshaking expansion (definition see Chapter 1). A closed system includes the program itself together with its environment description. A program with only SLHE processes is the simplest because no choices are involved, it is the parallel composition of a group of entirely sequential processes which communicate with each other. The modeling of the environment is the simplest too because the environment behaves deterministically like a group of entirely sequential processes. Furthermore, the whole system is perfectly symmetric (all the program processes and the environment processes are alike: all are straight-line handshaking expansions).

6.2 Study of an Example

Let's study a closed system with three SLHE processes P_1, P_2, P_3 communicating with each other via channels A, B, C. Each communication adopts a

certain four-phase handshaking protocol.

$$\begin{aligned} P_1 &\equiv *[ao\uparrow; [ai]; ao\downarrow; [\neg ai]; bo\uparrow; [bi]; bo\downarrow; [\neg bi]] \\ P_2 &\equiv *[[co]; ci\uparrow; [\neg co]; ci\downarrow; [bo]; bi\uparrow; [\neg bo]; bi\downarrow] \\ P_3 &\equiv *[[ao]; ai\uparrow; [\neg ao]; ai\downarrow; co\uparrow; [ci]; co\downarrow; [\neg ci]] \end{aligned}$$

Suppose the program consists of two processes $P_1 \parallel P_2$, and P_3 is the environment process. Now let's perform the reshuffling on the program in two different ways:

Case 1:
First let's postpone the downgoing part of communication on A in process P_1:

$$P_1' \equiv *[ao\uparrow; [ai]; bo\uparrow; [bi]; ao\downarrow; [\neg ai]; bo\downarrow; [\neg bi]]$$

This reshuffling causes a deadlock because communication on B can't get started without the completion of A.

Case 2:
Now let's postpone the downgoing part of communication on C in the process P_2 from the original closed system:

$$P_2' \equiv *[[co]; ci\uparrow; [bo]; bi\uparrow; [\neg co]; ci\downarrow; [\neg bo]; bi\downarrow]$$

The reshuffling doesn't cause any deadlock.

With the environment fixed, the two reshufflings on the same program have totally different outcomes. This simple introductory example leads us to the study of the interesting relationship between deadlocks and cycles in a communication dependency graph. The next section will be dedicated to the important concept of a communication dependency graph.

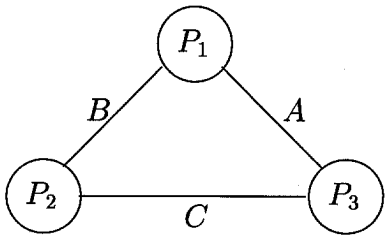

Figure 6.1: Configuration Diagram of Three Communicating Processes

6.3 Communication Dependency Graph

In a system consisting of communicating sequential processes, deadlocks are caused by communication failure, i.e. processes are blocked by waits on some of the input communication variables (as stated before, we don't allow other shared variables among processes). Due to this reason, in the liveness analysis we will ignore assignments on the local varaibles; in other words, we will project out the terms (transitions or waits) on local variables or state variables.

6.3.1 Projection Theorem

Def: Given a straight-line handshaking expansion with the set of program variables S, the *projection of the HSE on a subset of* S, say T, is a program derived from the original program by removing all transitions or waits on variables of $S \backslash T$. In other words, we *project on* the set T and *project out* the set $S \backslash T$.

For example, in the following handshaking expansion:

$$*[[li]; ro; [ri]; x\uparrow; [x]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; x\downarrow; [\neg x]; lo\downarrow]$$

where ri, ro, li, lo are communication variables and x is a state variable, the projection on the communication variables leaves us the handshaking expansion:

$$*[[li]; ro; [ri]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; lo\downarrow]$$

Theorem 6.3.1 (Projection Theorem) *Given a closed system with SLHE processes, the system is deadlock-free if and only if the projection of the system on the communication variables is deadlock-free.*

Proof: First, the projection of the whole system on the communication variables is well-defined: we project on each SLHE process of the communication variables. (Again no shared variables other than the communication variables are allowed between processes in this treatise.)

Necessity: We are going to show that if the original system is deadlock-free, so is the projected system. Suppose that the projected system has some deadlock, then it must be blocked by waits on some input communication variables. But the projection on the communication variables doesn't change any ordering among communication transitions. A block on some

wait $[xi]$ means the communication transition of type $xi\uparrow$ never happens in the projected system, therefore it never happens in the original system which contradicts the assumption of the original system being deadlock-free.

Sufficiency: Now we are going to show that if the projected system is deadlock-free, then the original system is deadlock-free. Suppose the original system has some deadlock, then some processes must be blocked by some waits. However, it won't be blocked by waits on any local variables, it can only be blocked on waits on some communication variables. Then the projected system has the same deadlock which leads to a contradiction. □

With the projection theorem, we can project out the local variables of a straight-line handshaking expansion and simplify the HSE to the one consisting only of communication actions. Then we will study the order relation among the communication transitions which is crucial to the liveness of a system.

6.3.2 Definition of Communication Dependency Relation

In Chapter 4, we studied the successor relation on transitions given the production rule set of a system. Here I will focus on the dependency relations on communication transitions (transitions on communication variables).

Def: The *dependency relations on communication transitions* are the successor relations on the transitions on the communication variables.

Therefore, the dependency relation is a partial order relation on the communication transitions. It is a subset of successor relations on the transitions on program variables.

Intracommunication Dependency Relation

Given a communication of certain communcation protocol on a port, we can establish dependency relations among the transitions on communication variables of the given communcation, I call such dependency relations the *intracommunication dependency relations.* For example, given the passive four-phase handshaking communication on L: $*[[li]; lo\uparrow; [\neg li]; lo\downarrow]$, we have the following (generating) set of the intracommunication dependency rela-

tions:

$$\begin{aligned}
\langle li\uparrow, k\rangle &\prec \langle lo\uparrow, k\rangle \\
\langle lo\uparrow, k\rangle &\prec \langle li\downarrow, k\rangle \\
\langle li\downarrow, k\rangle &\prec \langle lo\downarrow, k\rangle \\
\langle lo\downarrow, k\rangle &\prec \langle li\uparrow, k+1\rangle
\end{aligned}$$

Note: In most cases, we are interested in the generating set of a partially ordered set. For simplicity, we omit the word "generating set" when talking about the dependency relations.
(end of Note)

In a similar manner we have the intracommunication dependency relations for the active four-phase handshaking communication on L:
$*[lo\uparrow; [li]; lo\downarrow; [\neg li]]$

$$\begin{aligned}
\langle lo\uparrow, k\rangle &\prec \langle li\uparrow, k\rangle \\
\langle li\uparrow, k\rangle &\prec \langle lo\downarrow, k\rangle \\
\langle lo\downarrow, k\rangle &\prec \langle li\downarrow, k\rangle \\
\langle li\downarrow, k\rangle &\prec \langle lo\uparrow, k+1\rangle
\end{aligned}$$

and for the lazy-active four-phase handshaking communication on L:
$*[[\neg li]; lo\uparrow; [li]; lo\downarrow]$

$$\begin{aligned}
\langle li\downarrow, k-1\rangle &\prec \langle lo\uparrow, k\rangle \\
\langle lo\uparrow, k\rangle &\prec \langle li\uparrow, k\rangle \\
\langle li\uparrow, k\rangle &\prec \langle lo\downarrow, k\rangle \\
\langle lo\downarrow, k\rangle &\prec \langle li\downarrow, k\rangle
\end{aligned}$$

So the active and the lazy-active four-phase handshaking protocol share the same intracommunication dependency relations. Replacing an active four-phase communication with its lazy-active counterpart plays an important role in the improvement of the performance (especially efficiency) of the system. This operation can be regarded as a special case of reshuffling: we postpone the wait on the downgoing transition of the input variable, i.e. $[\neg li]$, till the begining of the next communication action. Later, we will give a proof showing why this reshuffling is always deadlock-free.

It is easy to check that the aforementioned dependency relation sets for different communication protocols are all complete and irreducible.

Intercommunication Dependency Relation

Now I will study the dependency relations among communications on different ports, which were not covered in the previous section. I call this set of dependency relations the *intercommunication dependency relations*, denoted as W. Let S be the complete set of dependency relations on communication transitions, and let V be the set of intracommunication dependecny relations, then $W = S \backslash V$.

The intercommunication dependency relation can become very complicated due to the number of communications that interact with one another and the ways in which they interact. Even in the simplist case when communications are given on two ports, the dependency relations between them could evolve into many variations. Let's study a few common cases:

Case 1: Sequential composition of communications on ports L and R:

$$*[lo\uparrow; [li]; lo\downarrow; [\neg li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]]$$

the intercommunication dependency relations between the two can be reduced to the following:

$$\begin{aligned} \langle li\downarrow, k\rangle &\prec \langle ro\uparrow, k\rangle \\ \langle ri\downarrow, k\rangle &\prec \langle lo\uparrow, k+1\rangle \end{aligned}$$

Case 2: Interleaving between the two communications, which could be the result of reshuffling:

$$*[lo\uparrow; [li]; ro\uparrow; [ri]; lo\downarrow; [\neg li]; ro\downarrow; [\neg ri]]$$

then the irreducible set of intercommunication dependency relations are:

$$\begin{aligned} \langle li\uparrow, k\rangle &\prec \langle ro\uparrow, k\rangle \\ \langle ri\uparrow, k\rangle &\prec \langle lo\downarrow, k\rangle \\ \langle li\downarrow, k\rangle &\prec \langle ro\downarrow, k\rangle \\ \langle ri\downarrow, k\rangle &\prec \langle lo\uparrow, k+1\rangle \end{aligned}$$

Case 3: Parallel composition of two communications (in a HSE): $L \parallel R$, there is no intercommunication dependency relation between the two.

Note: Here I abuse the term *communication* a little bit: when I say two communications I mean communications on two ports. A communication is

different from communication action in this treatise, for instance, $L; L$ is an example of two communication actions but is the same communication on L. (end of Note)

The dependency relations I listed above are the immediate ones. In many cases, two communication actions are related indirectly. For example in $L; S; R$, communication actions L and R are related sequentially via S. However, by using transitivity we can derive the indirect dependency relations through the partial order chain $L \prec S$ and $S \prec R$. Therefore we won't "lose" any dependency relations by restricting consideration to the immediate orderings.

The intercommunication dependency relations listed above are irreducible. Though the union of the intercommunication dependency relations and intracommunication dependency relations generates the entire set of dependency relations for the communication actions, yet the intercommunication and intracommunication dependency relations are not necessarily disjoint from one another, and therefore their union is usually not irreducible. For example in case 2, the union of the two gives a reducible subset of partial orders.

Since each communication links at least two processes and in the above examples we derive the dependency relations from one side of the communication, conceivably the dependency relations could be derived in the same way from the other side of the communication. Therefore, correctness requires consistency between the separately-derived dependency relations, which is the major topic of this chapter.

6.3.3 Definition of Communication Dependency Graph

Def: A *communication dependency graph* is a graphic representation of generating set of communication dependency relations in a closed system . In a communication dependency graph (V, E), each node $v \in V$ represents a communication transition, and each edge (v_s, v_e) represents the successor relation $v_s \prec v_e$ (graphically it is a directed edge pointing from v_s to v_e).

6.3.4 Properties of Communication Dependency Graph

Def: Let $v_1, v_2, \ldots, v_n$ be nodes in the CDG (V, E), and $(v_i, v_{i+1})$with $i =$

$1, \ldots, n-1$ be edges in the CDG, then $(v_1, v_2, \ldots, v_n)$ forms a *path* in the CDG and we say v_n is *reachable* from v_1.

Property 6.3.1 *If there is a path from node v_1 to node v_n, then this corresponds to the successor relation $v_1 \prec v_n$.*

Proof: Trivial from the definition of the successor relation and the CDG. □

Def: A node in the CDG that is not reachable from any other node is a *root.* A node in the CDG with no successors is a *leaf.*

Property 6.3.2 *In a CDG, for each edge (v_s, v_e), v_e must be a successor of v_s.*

Proof: By definition, the CDG is a graphic representation of the generating set for a partially ordered set. Therefore, there is a one-to-one correspondence between an edge in CDG and a partial order relation in the generating set. Suppose there exists in the CDG an edge (v_1, v_n) such that v_n is not the successor of v_1, then there must exists $v_2, \ldots, v_{n-1}$, $(n \geq 3)$ with such that v_{i+1} is the successor of v_i for $1 \leq i \leq n-1$. But $v_i \prec v_{i+1}$ must be in the generating set, therefore (v_i, v_{i+1}) is in the CDG, which implies that v_n is reachable from v_1 via $v_2, \ldots, v_{n-1}$. Thus (v_1, v_n) is not an edge in the CDG, contradiction. □

Corollary 6.3.1 *For each edge in CDG (v_s, v_e), there exists no path other than (v_s, v_e) starting from v_s and ending at v_e.*

Between two nodes in CDG, there can be more than one path. For example, the CDG for the following generating set:

$$a \prec b \prec c \prec d$$

$$a \prec e \prec d$$

between nodes a and d there are two paths.

6.4 Cycles in Communication Dependency Graph

A communication dependency graph is a directed graph. This section will study the meaning of cycles in a CDG.

Def: A *cycle* is a path $(v_1, v_2, \ldots, v_n, v_1)$ in CDG with $n \geq 2$. If the cycle also satisfies $v_i \neq v_j$ for $i \neq j$, then it is a *simple cycle.*

Corollary 6.4.1 *Any cycle must includes a simple cycle.*

Proof: Suppose the cycle $(v_1, v_2, \ldots, v_n, v_1)$ is not a simple cycle, then there exists $1 \leq i < j \leq n$ such that $v_i = v_j$. Now we have a smaller subcycle $(v_i, \ldots, v_j)$: if it is a simple cycle we are done; if it is not, then repeat the procedure to find a smaller cycle. Since there are finitely many nodes in the original cycle, we will finally find the smallest cycle $(v_a, \ldots, v_b)$ such that for any $a \leq s < t \leq b-1$ we have $v_s \neq v_t$ (by definition of the smallest cycle). So $(v_a, \ldots, v_b)$ is a simple cycle. □

Lemma 6.4.2 *Any CDG which represents the generating set of a partially ordered set is acyclic.*

Proof: Suppose there is a cycle in the CDG, then from the previous corollary we have a simple cycle $(v_1, v_2, \ldots, v_n, v_1)$ with $v_1 \prec v_2 \prec \ldots \prec v_n \prec v_1$. Therefore $v_1 \prec v_1$ which violates the anti-reflexity. □

Note: In a partially ordered set, the relations $x \preceq y$ and $x \neq y$ are equivalent to $x \prec y$, therefore in the CDG an edge (v_s, v_e) satisfies $v_s \prec v_e$ since $v_s \neq v_e$. (end of Note)

6.5 Deadlock and Cycles in CDG: An Example

In the beginning of the Chapter, I studied an example of the reshuffling of three processes. In one case the reshuffling causes a deadlock, while in the other cases the reshuffling is deadlock-free. Now let's build the CDG for the original HSE and the reshuffled ones to illustrate the relationship between

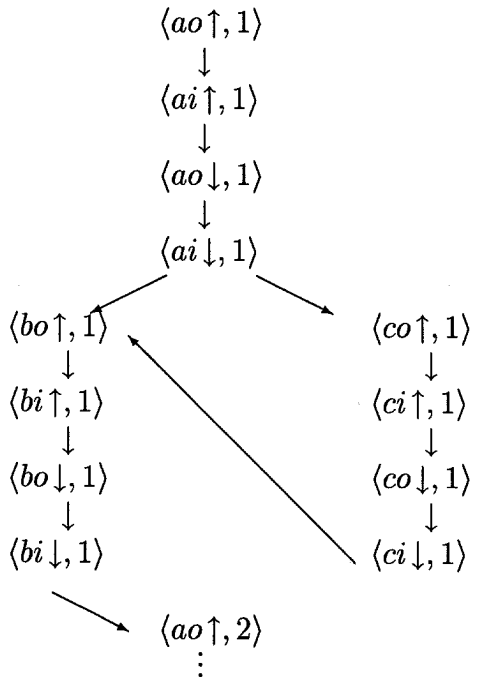

Figure 6.2: CDG for the Original HSE of Three Processes

cycles in the CDG and deadlock.

The HSE for the original closed system is:

$$\begin{aligned} P_1 &\equiv *[ao\uparrow; [ai]; ao\downarrow; [\neg ai]; bo\uparrow; [bi]; bo\downarrow; [\neg bi]] \\ P_2 &\equiv *[[co]; ci\uparrow; [\neg co]; ci\downarrow; [bo]; bi\uparrow; [\neg bo]; bi\downarrow] \\ P_3 &\equiv *[[ao]; ai\uparrow; [\neg ao]; ai\downarrow; co\uparrow; [ci]; co\downarrow; [\neg ci]] \end{aligned}$$

and the corresponding communication dependency graph is showed in figure 6.2 on page 39.

For the first reshuffling where P_1 is reshuffled to:

$$P_1' \equiv *[ao\uparrow; [ai]; bo\uparrow; [bi]; ao\downarrow; [\neg ai]; bo\downarrow; [\neg bi]]$$

while P_2, P_3 remain the same:

$$\begin{aligned} P_2 &\equiv *[[co]; ci\uparrow; [\neg co]; ci\downarrow; [bo]; bi\uparrow; [\neg bo]; bi\downarrow] \\ P_3 &\equiv *[[ao]; ai\uparrow; [\neg ao]; ai\downarrow; co\uparrow; [ci]; co\downarrow; [\neg ci]] \end{aligned}$$

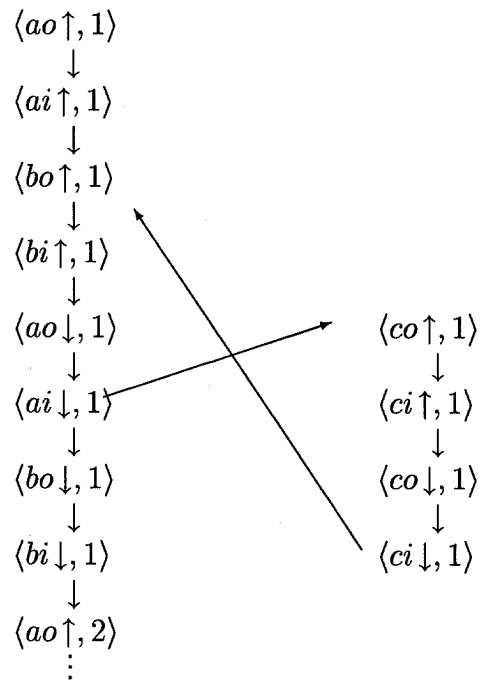

Figure 6.3: CDG for the Reshuffled HSE (with Deadlock) of Three Processes

the corresponding communication dependency graph is shown in figure 6.3 on page 40.

For the second reshuffling where P_2 is reshuffled to:

$$P_2' \equiv *[[co]; ci\uparrow; [bo]; bi\uparrow; [\neg co]; ci\downarrow; [\neg bo]; bi\downarrow]$$

while P_1, P_3 remain the same:

$$\begin{aligned} P_1 &\equiv *[ao\uparrow; [ai]; ao\downarrow; [\neg ai]; bo\uparrow; [bi]; bo\downarrow; [\neg bi]] \\ P_3 &\equiv *[[ao]; ai\uparrow; [\neg ao]; ai\downarrow; co\uparrow; [ci]; co\downarrow; [\neg ci]] \end{aligned}$$

the corresponding communication dependency graph is shown in figure 6.4 on page 41.

In the CDG for the original HSE, there are no cycles; also the CDG for the second deadlock-free reshuffling has no cycles. But for the first reshuffling, which introduces deadlock, its CDG contains the cycle ($\langle bo\uparrow, 1\rangle, \langle bi\uparrow$

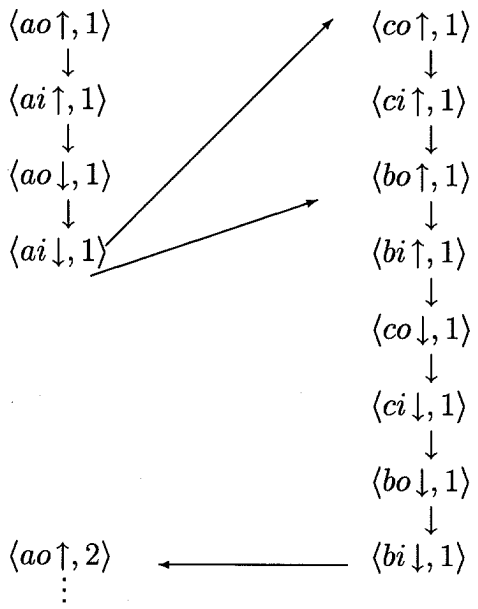

Figure 6.4: CDG for the Reshuffled HSE (Deadlock-free) of Three Processes

, 1⟩, ⟨ao ↓, 1⟩, ⟨ai ↓, 1⟩, ⟨co ↑, 1⟩, ⟨ci ↑, 1⟩, ⟨co ↓, 1⟩, ⟨ci ↓, 1⟩, ⟨bo ↑, 1⟩) (see figure 6.3). Therefore, it illustrates the relation between cycles in the CDG of a system and deadlock.

6.6 Deadlock and Cycles in CDG: Theory

In the aforementioned example, it is a trivial and easy job to build the CDG for the whole closed system. However, we are not always this lucky; on the contrary, in most cases when the system involving choices and the environment is complex, not only "know-how" becomes a problem but also the construction of the CDG becomes extremely complicated.

Another nice feature of the CDG in the previous example is that the CDG is unique for the system (reshuffled or unreshuffled). Unfortunately, this feature is related to many factors: mainly the program itself and the description of the environment etc.

However, the good news here is that for a system with only SLHE processes, neither of the aforementioned difficulties is a problem: there is a unique CDG related to the system and the construction of the CDG is easy and straightforward. In order to stick to our main task here – detecting deadlock and providing motivation for the reader, I will first give some preliminary results on cycles in the CDG and deadlock, before going on with the construction of the CDG and the proof of its uniqueness in a closed system with only SLHE processes. For now assume that for any closed sytem with only SLHE processes there is a unique CDG attached to it.

6.6.1 Unique Computation

Communication dependency relations are defined on transitions on communication variables, therefore they are related to the actual execution of the circuit which implements the system. Two executions are considered to be in the same computation if they share the same successor relations.

For example, consider the two executions for the C-element:

$$\langle x\uparrow,1\rangle;\langle y\uparrow,1\rangle;\langle z\uparrow,1\rangle;\langle x\downarrow,1\rangle;\langle y\downarrow,1\rangle;\langle z\downarrow,1\rangle;\langle x\uparrow,2\rangle;\langle y\uparrow,2\rangle;\langle z\uparrow,2\rangle;\ldots$$

and

$$\langle y\uparrow,1\rangle;\langle x\uparrow,1\rangle;\langle z\uparrow,1\rangle;\langle y\downarrow,1\rangle;\langle x\downarrow,1\rangle;\langle z\downarrow,1\rangle;\langle y\uparrow,2\rangle;\langle x\uparrow,2\rangle;\langle z\uparrow,2\rangle;\ldots$$

they share the same successor relation:

$$\begin{aligned}
\langle x\uparrow,i\rangle &\prec \langle z\uparrow,i\rangle\\
\langle y\uparrow,i\rangle &\prec \langle z\uparrow,i\rangle\\
\langle z\uparrow,i\rangle &\prec \langle x\downarrow,i\rangle\\
\langle z\uparrow,i\rangle &\prec \langle y\downarrow,i\rangle\\
\langle x\downarrow,i\rangle &\prec \langle z\downarrow,i\rangle\\
\langle y\downarrow,i\rangle &\prec \langle z\downarrow,i\rangle\\
\langle z\downarrow,i\rangle &\prec \langle x\uparrow,i+1\rangle\\
\langle z\downarrow,i\rangle &\prec \langle y\uparrow,i+1\rangle
\end{aligned}$$

Theorem 6.6.1 *Any closed system with only SLHE processes has a unique computation.*

Proof: A computation is a set of successor relations on transitions. For a closed system with only SLHE processes, because the set of transitions is trivial and each transition has a unique cause set, the successor relations are therefore unique. □

Note: Here there is no problem of "multi-transitions" in an SLHE. Because in terms of transitions they are different and syntactically may be identified by renaming.
(end of Note)

6.6.2 Deadlock in a Closed SLHE System

Def: A *configuration graph* is an undirected graph which represents the communication connectivity between processes. Each node represents a process, each edge represents a channel connecting two processes (for simplicity, I exclude the multi-sender and multi-receiver channels). A configuration graph is *complete* if there are no dangling edges. A configuration graph is *connected* if between any two nodes there is a path.

Def: A *connected closed system* refers to a set of processes whose configuration graph is complete and connected.

Lemma 6.6.1 *Given a connected closed system, if one SLHE process is blocked by some waits then the whole system will have a deadlock.*

Proof: If one process is blocked by some wait this means that the other process never gets to execute that transition which makes the wait true, then that process must be blocked at some other wait. And these two processes must have connected with processes other than the two processes. Suppose process C is connected with processes A and B, then C must be blocked at some wait for communication from A or B because eventually A or B will halt. By induction some process D connected with A, B or C will be blocked, and eventually the whole system will be deadlocked. □

6.6.3 Relation between Deadlock and Cycles in CDG

For a system that is not connected, we can physically break it into several connected parts, each subsystem forms a connected closed system. Therefore from now on, the closed systems I refer to are all connected closed systems.

Theorem 6.6.2 *Given a closed system with only SLHE processes, the system is deadlock-free if and only if its corresponding CDG (Communication Dependency Graph) is acyclic.*

Proof:

First, it is well-defined to say that "the system is deadlock-free": Absence of deadlock is a property of a computation, but a closed system with only SLHE processes has a unique computation therefore it is a system property for this special type of system.

Sufficiency: We are going to show that if the CDG is acyclic then the system is deadlock-free. Proof by contradiction: suppose the system has some deadlock, then its only computation has the deadlock, which means all the executions will be deadlocked. If the CDG is acyclic then there is an execution given by the following timing function [1]:

$$t(f) = \begin{cases} 0 & \text{if } f \text{ is a root} \\ \max\{t(e) + \alpha | e \overset{\alpha}{\mapsto} f \in E\} & \text{otherwise} \end{cases}$$

Necessity: Let the cycle be (a, b, c, a), where a, b, c are communication transitions, then we know communication transition b can fire only after a fires and c can fire only after b fires, so by transitivity c can fire only after a fires; but from the CDG a fires after c fires. So a will never be fired, therefore some wait on a will be finally blocked in some SLHE process and this will eventually cause the system deadlock. □

6.7 CDG of a Closed System with SLHE Processes

6.7.1 Construction of the CDG for a Single SLHE Process

Suppose each SLHE processes $*[S_1]$ assumes the standard format in which waits and transitions alternate in S_1. It is easy to standardize an SLHE process into this format:

- Replace the consecutive waits $[x_0]; [x_1]$ with the single wait $[x_0 \wedge x_1]$.
- Separate the consecutive transitions $x\updownarrow; y\updownarrow$ with $x\updownarrow; [\mathbf{true}]; y\updownarrow$.

The set of nodes of the CDG, V, includes all the transitions on the communication variables in the SLHE process. The set of edges of the CDG, E, includes the following:

- When $[x_1 \wedge \ldots \wedge x_n]; y$ then $(trans(x_i), y)$ is in E for $1 \leq i \leq n$.
- When $y; \ldots; [x_1 \wedge \ldots \wedge x_n]$ then $(y, trans(x_k))$ is in E if $[x_k]$ and y are the wait and transition of some communication on the same port.
- When $y; [x_1 \wedge \ldots \wedge x_n]; z$ then (y, z) is in E if none of the $[x_k]$ and y are the wait and transition of some communication on the same port. In the special case: $y; [\mathbf{true}]; z$ then (y, z) is in E.

For example, the HSE for the control of a one-place buffer:

$$*[[li]; lo\uparrow; [\neg li]; lo\downarrow; ro\uparrow; [ri]; ro\downarrow; [\neg ri]]$$

is standardized to:

$$*[[li \wedge \neg ri^{-1}]; lo\uparrow; [\neg li]; lo\downarrow; [\mathbf{true}]; ro\uparrow; [ri]; ro\downarrow]$$

Then using the above rules for construction we get the following CDG :

$$\begin{aligned} V &= \{li\uparrow, li\downarrow, lo\uparrow, lo\downarrow, ri\uparrow, ri\downarrow, ro\uparrow, ro\downarrow\} \\ E &= \{(\langle li\uparrow, i\rangle, \langle lo\uparrow, i\rangle), (\langle ri\downarrow, i-1\rangle, \langle lo\uparrow, i\rangle) \\ &\quad (\langle lo\uparrow, i\rangle, \langle li\downarrow, i\rangle), (\langle li\downarrow, i\rangle, \langle lo\downarrow, i\rangle), \\ &\quad (\langle lo\downarrow, i\rangle, \langle ro\uparrow, i\rangle), (\langle ro\uparrow, i\rangle, \langle ri\uparrow, i\rangle), \\ &\quad (\langle ri\uparrow, i\rangle, \langle ro\downarrow, i\rangle), (\langle ro\downarrow, i\rangle, \langle ri\downarrow, i\rangle)\} \end{aligned}$$

Note: The standardization procedure I have talked about above treats the HSE as an infinite unwrapped program. However, in the given example I did the standardization on a repitition program, and in that case I should be a little more careful because usually the transition x there stands for $\langle x, i-\epsilon\rangle$, where the ϵ is some nonnegative number.
(end of Note)

Now we need to prove that the directed graph built above is the communication dependency graph for the SLHE process.

Theorem 6.7.1 *The directed graph built using the rules is complete and irreducible for the communication dependency relations, and thus represents the unique generating set. Equivalently, it is the communication dependency graph for the SLHE.*

Proof: First, I will show that set E built using the rules can generate the entire set of communication dependency relations. We know that the set of dependency relations, including all the successors which can be built from the set of PR rules, can generate the entire set. For any PR rule on an output variable z, we assume $ai \wedge bo \mapsto z\uparrow$. If $bo\uparrow$ doesn't precede $z\uparrow$, there must be a transition between $bo\uparrow$ and $z\uparrow$. By induction, finally we can build a chain of transitions $bo\uparrow \prec t_0 \prec \ldots \prec t_k \prec z\uparrow$. If $[ai]$ doesn't precede $z\uparrow$, it must precede some transition t before the $z\uparrow$ if the SLHE is in standard form. But there is a path from that transition t to $z\uparrow$ as argued before. So we have $ai \prec t \prec z\uparrow$ which gives finally $ai \prec z\uparrow$. For any PR rule on an input variable u, we have $ci \wedge bo \longrightarrow u$, so the derived ER rule system is complete. Notice that what we care about is the partial ordering between the transitions, therefore from the complete set of rules we can derive all the partial orders.

It is irreducible. Suppose not, then there exists an edge (v_s, v_e) such that there is a path of length greater than 2 from v_s to v_e. But the edges are drawn according to the aforementioned three cases, for cases 1 and 3 it is obvious that there is no chain of transitions inbetween, for case 2, if there are chains of transitions in between, then there must be a transition t such that $yo\updownarrow \prec t \prec yi\updownarrow$, then t must be in the same communication as yi, but $yo\updownarrow$ is the closest one, contradiction.

So we have proven that the partial relation set is minimal. □

6.7.2 Construction of the CDG for a Closed System

From the previous section, we can build the CDG for any SLHE process. Now assume that we have a closed system consisting of a finite number of SLHE processes, then the CDG for the system can be built by following these steps:

- First build the CDG for each SLHE process.
- Then merge the nodes which represent the same transition.
- Remove an edge if there exists a path of length greater than 2 between its nodes.

Theorem 6.7.2 *The graph is complete and irreducible, therefore it is the CDG for the whole system.*

Proof: The processes are composed parallelly, the union of the CDGs for each process will generate the whole system's communication dependency relations.

It is irreducible because for each edge there is no path of greater length. □

So far I have established the theoretical construction of the communication dependency graph for any closed system with SLHE processes and the equivalence between proving that the system is deadlock-free and showing that the corresponding CDG is acyclic.

However the system is non-terminating with an infinite number of transitions, therefore the CDG involves an infinite number of nodes and edges which makes the analysis practically impossible. So the next step is to reduce this analysis with an infinite nature to an equivalent analysis with a finite nature.

6.8 Repetitive Systems

First consider the following closed systems $*[S_1] \parallel *[S_2]$ and $*[S_1 \parallel S_2]$. These two systems are syntactically different because the second system requires the completion point of both S_1 and S_2, while the first system has no such requirement. The second system is interesting for the liveness analysis because if we can show that $S_1 \parallel S_2$ is deadlock-free, then $*[S_1 \parallel S_2]$ is deadlock-free too. Proving that a system with an infinite number of transitions, such as $*[S_1 \parallel S_2]$, is deadlock-free reduces to the task of proving that a system with a finite number of transitions, $S_1 \parallel S_2$, is deadlock-free. The second system, $*[S_1 \parallel S_2]$, belongs to a special group of systems which I call repetitive systems.

Def: A *repetitive system* is a system that has the following operational behavior:

- Each execution of the program is a repetitive concatenation of an execution of the main program body P (note there can be numerous executions of P also). Semantically, it can be written as $*[P]$.
- If the system is in the initial state before executing the program body P, then after any terminiating execution of P the system will end up at the initial state, i.e. $\{\sigma_{init}\}P\{\sigma_{init}\}$.

Theorem 6.8.1 *A repetitive system $*[P]$ is deadlock-free if and only if P is deadlock-free.*

Proof: Deadlock is a special program state $\perp$ which is observed in a non-terminating execution of the program. If P is deadlock-free, then it means that all the executions of P terminate and that $\{\sigma_{init}\}P\{\sigma_{init}\}$ always holds.

Sufficiency (the "if" part) : If $*[P]$ is not deadlock-free, then there exists a nonterminating exection of $*[P]$ such that $\{\sigma_{init}\} * [P]\{\perp\}$ and $\perp$ could be observed in some finite time. Therefore, n exists such that $\{\sigma_{init}\} \underbrace{P;\ldots;P}_{n}\{\perp$ }. Using inducition and the second property $\{\sigma_{init}\}P\{\sigma_{init}\}$, we have $\{\sigma_{init}\}P\{\sigma_{init}\} \underbrace{P;\ldots;P}_{n-1}\{\perp\}$. Finally we get $\{\sigma_{init}\}P\{\perp\}$, contradicting that P is deadlock-free.

Necessity (the "only if" part): Easy. Suppose P is not deadlock-free, then there exists an execution of P such that $\{\sigma_{init}\}P\{\perp\}$. Then the same execution could happen to $*[P]$ and leaves the program $\{\sigma_{init}\} * [P]\{\perp\}$ in a deadlock state, which contradicts the assumption that $*[P]$ be deadlock-free. □

Syntactically a repetitive system can be written as $*[P]$ where P is a finite program body. A simple example of this is $Sys_1 \equiv *[(A;B) \parallel (A';C) \parallel (B';C')]$ where $(A,A'),(B,B'),(C,C')$ are pairs of communications on channels A,B,C. However, from the engineering point of view this program could be inefficient compared with $Sys_2 \equiv *[A;B] \parallel *[A';C] \parallel *[B';C']$, because the former one requires the rendezvous of the completion point of $(A;B),(A';C)$ and $(B';C')$. However, Sys_1 can be considered as a *refinement* of Sys_2 in that the set of executions of Sys_1 is a subset of the executions of Sys_2. In fact, we have the following theorem:

Theorem 6.8.2 *Given two systems Sys_1 and Sys_2, and Sys_1 is a refinement of Sys_2, if Sys_2 is deadlock-free then Sys_1 is deadlock-free.*

Proof: Sys_2 is deadlock-free means that all of its executions are absent of deadlock. Since Sys_1 is an refinement of Sys_2, all the executions of Sys_1 are a subset of Sys_2. Therefore all the execution of Sys_1 are absent of deadlock. So Sys_1 is deadlock-free. □

Corollary 6.8.1 *Let $S_1, \ldots, S_n$ be SLHE, if $\{\sigma_{init}\}S_1, \ldots, S_n\{\sigma_{init}\}$ holds then $*[S_1 \parallel \ldots \parallel S_n]$ is a repetitive system and $*[S_1 \parallel \ldots \parallel S_n]$ is a refinement of $*[S_1] \parallel \ldots \parallel *[S_n]$. If $*[S_1] \parallel \ldots \parallel *[S_n]$ is deadlock-free, then so is $*[S_1 \parallel \ldots \parallel S_n]$.*

Therefore, we have the following relationship:

$$*[S_1] \parallel \ldots \parallel *[S_n] \text{ deadlock free} \Rightarrow *[S_1 \parallel \ldots \parallel S_n] \text{ deadlock-free}$$
$$\Leftrightarrow S_1 \parallel \ldots \parallel S_n \text{ deadlock-free}$$

Therefore if we can prove the following theorem then we can prove $*[S_1] \parallel \ldots \parallel *[S_n]$ is deadlock-free by showing $S_1 \parallel \ldots \parallel S_n$ deadlock-free for closed systems with SLHE processes.

Theorem 6.8.3 *$S_1, \ldots, S_n$ are all straight-line handshaking expansions. Let $Sys_1 \equiv *[S_1 \parallel \ldots \parallel S_n]$ be a refinement of $Sys_1 \equiv *[S_1] \parallel \ldots \parallel *[S_n]$. If $*[S_1 \parallel \ldots \parallel S_n]$ is deadlock-free, then $*[S_1] \parallel \ldots *[S_n]$ is deadlock-free.*

Before proving the above theorem, first I need to define the concatenation of two paths:

Def: Given two paths $(v_1, \ldots, v_m)$ and $(v_m, \ldots, v_n)$ where the end node of the first path is the beginnig node of the second path, the concatenation of the two paths $(v_1, \ldots, v_m)$++$(v_m, \ldots, v_n)$ is the path $(v_1, \ldots, v_{m-1}, v_m, v_{m+1}, \ldots, v_n)$.

Proof: (of theorem 6.8.3) by contradiction. Suppose that $Sys_2 \equiv *[S_1] \parallel \ldots \parallel *[S_n]$ is not deadlock-free, then there exists a cycle $(v_1, \ldots, v_n, v_1)$ in the corresponding CDG of Sys_2. But $Sys_1 \equiv *[S_1 \parallel \ldots \parallel S_n]$ is a refinement of Sys_2, and so Sys_1 contains more successor relations. Therefore, for the cycle $(v_1, \ldots, v_n, v_1)$ in the CDG of Sys_2, we can find a cycle in the CDG of Sys_2 by concatenating the paths between nodes v_i and v_{i+1}:

$$(path(v_1, v_2)++path(v_2, v_3)++\ldots++path(v_{n-1}, v_n)++path(v_n, v_1))$$

which implies that Sys_1 is not deadlock-free. Contradiction to the assumption. □

So for any closed system with only SLHE processes:

$$*[S_1] \parallel \ldots \parallel *[S_n]$$

if $*[S_1 \parallel \ldots \parallel S_n]$ is a repetitive program then $*[S_1] \parallel \ldots \parallel *[S_n]$ is deadlock-free if and only if:

$$S_1 \parallel \ldots \parallel S_n$$

is deadlock-free.

6.9 Deadlock Detection using Repetitive System

This section I will construct an algorithm for building the repetitive refinement of a closed system with only SLHE processes: $*[S_1] \parallel \ldots \parallel *[S_n]$.

6.9.1 An Example

Consider the example of the two-to-four phase converter:

$$\begin{aligned} \mathit{Conv} &\equiv *[[li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\downarrow] \\ \text{2-}\mathit{phase} &\equiv *[li\uparrow; [lo]; li\downarrow; [\neg lo]] \\ \text{4-}\mathit{phase} &\equiv *[[ro]; ri\uparrow; [\neg ro]; ri\downarrow] \end{aligned}$$

The closed system consists of $\mathit{Conv} \parallel \text{2-}\mathit{phase} \parallel \text{4-}\mathit{phase}$.

The repetitive refinement of the above the system is :

$$\begin{aligned} *[\ & [li]; ro^0\uparrow; [ri^0]; ro^0\downarrow; [\neg ri^0]; lo\uparrow; [\neg li]; ro^1\uparrow; [ri^1]; ro^1\downarrow; [\neg ri^1]; lo\downarrow \\ \parallel\ & li\uparrow; [lo]; li\downarrow; [\neg lo] \\ \parallel\ & [ro^0]; ri^0\uparrow; [\neg ro^0]; ri^0\downarrow; [ro^1]; ri^1\uparrow; [\neg ro^1]; ri^1\downarrow \\ ]\ & \end{aligned}$$

where the transition $\langle ro^0\updownarrow, i\rangle$ on the variable ro corresponds to the transition $\langle ro\updownarrow, 2i-1\rangle$. Similarly, $\langle ri^0\updownarrow, i\rangle$ on the variable ri corresponds to the transition $\langle ri\updownarrow, 2i-1\rangle$; $\langle ro^1\updownarrow, i\rangle$ on the variable ro corresponds to the transition $\langle ro\updownarrow, 2i\rangle$; $\langle ri^1\updownarrow, i\rangle$ on the variable ri corresponds to the transition $\langle ri\updownarrow, 2i\rangle$.

6.9.2 Building a Repetitive System: Theory

Again all the systems studied here are closed systems with only SLHE processes. Furthermore, I assume each process contain only communications which can be achieved by projection on only communication variables.

Def: Given a system with the SLHE processes: $*[S_1] \parallel \ldots \parallel *[S_n]$, suppose process $*[S_i]$ communicates with process $*[S_j]$ via channel X, if the number of communications on X in S_i equals the number of communications on X in S_j, then we say that the communications on X is *well-balanced.*

Theorem 6.9.1 *The system* $Sys_2 \equiv *[S_1 \parallel \ldots \parallel S_n]$ *is a refined repetitive counterpart of system* $Sys_1 \equiv *[S_1] \parallel \ldots \parallel *[S_n]$ *if all the communications in* Sys_1 *are well-balanced.*

Proof: If all the communications in Sys_1 are well-balanced, then $S_1 \parallel \ldots \parallel S_n$ can have an coincident completion point, and when the program starts from its initial state σ_{init}, then $\{\sigma_{init}\}S_1 \parallel \ldots \parallel S_n\{\sigma_{init}\}$ holds. Therefore the refinement $*[S_1 \parallel \ldots \parallel S_n]$ forms a repetitive system. □

Def: A SLHE process $*[S_1]$ is in its *minimal form* if S_1 can not be written as $\underbrace{s;\ldots;s}_{n}$ where n is some integer greater than 1 and s is a segment of S_1.

For example, the minimal form of $*[[li]; lo\uparrow; [\neg li]; lo\downarrow; [li]; lo\uparrow; [\neg li]; lo\downarrow]$ is $*[[li]; lo\uparrow; [\neg li]; lo\downarrow]$.

Theorem 6.9.2 *Given a SLHE system* $*[S_1] \parallel *[S_2]$ *where both* S_1, S_2 *are in their minimal form, then there exists unique* l_1, l_2 *such that* $\gcd(l_1, l_2) = 1$, *and* $*[(S_1)^{l_1}] \parallel *[(S_2)^{l_2}]$ *has all its communications well-balanced. Here* $(S_i)^{l_i}$ *represents* $\underbrace{S_i;\ldots;S_i}_{l_i}$. *Therefore, the refined repetitive counterpart is* $*[(S_1)^{l_1} \parallel (S_2)^{l_2}]$.

Proof: It is easy to balance one communication between two processes. For example suppose X is well-balanced in $*[(S_1)^{k_1}] \parallel *[(S_2)^{k_2}]$ and $\gcd(k_1, k_2) = 1$. Let $n(X)$ be the number of communications X in S_1 and $m(X)$ be the number of communications X in S_2. Then we have $k_1 n(X) = k_2 m(X)$. Now pick another communication Y, if Y is not well-balanced in $*[(S_1)^{k_1}] \parallel *[(S_2)^{k_2}]$, then at the $k_1 n(X)$ *th* completion of the communication X, Y will be completed $k_1 n(Y)$ times in $*[(S_1)^{k_1}]$ and $k_2 m(Y)$ times in $*[(S_2)^{k_2}]$ (where $k_1 n(Y) \neq k_2 m(Y)$). This is impossible because the number of completions of Y is different in the two processes which violates the synchronization condition. Therefore Y must be balanced in $*[(S_1)^{k_1}] \parallel *[(S_2)^{k_2}]$. And we can have $l_1 = k_1,\ l_2 = k_2$. □

Theorem 6.9.3 *Given any system with only SLHE processes,* $*[S_1] \parallel \ldots \parallel *[S_n]$ *where all the* S_i *are in their minimal form, then there exists unique* $l_1, \ldots, l_n$ *such that* $\gcd(l_1, \ldots, l_n) = 1$, *and* $*[(S_1)^{l_1}] \parallel \ldots \parallel *[(S_n)^{l_n}]$ *has all its communications well-balanced. Here* $(S_i)^{l_i}$ *represents* $\underbrace{S_i; \ldots; S_i}_{l_i}$. *Therefore, the refined repetitive counterpart is* $*[(S_1)^{l_1} \parallel \ldots \parallel (S_n)^{l_n}]$.

The proof of the theorem depends on the n-tuple $(l_1, \ldots, l_n)$, therefore I will first give the algorithm for constructing the n-tuple and then prove the theorem.

6.9.3 Building a Repetitive System: Algorithm

In this section, I will give an algorithm to balance the communications in a closed system with SLHE processes: $*[S_1] \parallel \ldots \parallel *[S_n]$. Let's first start with the simple case – systems with only two SLHE processes.

Balancing a System with Two SLHE Processes

Given a closed system with two SLHE processes: $*[S_1] \parallel *[S_2]$ with S_1, S_2 both in their minimal form, pick a communication A between the two processes. Let $m(A)$ be the number of A communications in process $*[S_1]$ and $n(A)$ be the number of A communications in process $*[S_2]$, then we have:

$$k_1 = \frac{lcm(n(A), m(A))}{n(A)} \quad \forall \text{ communications } A$$

$$k_2 = \frac{lcm(n(A), m(A))}{m(A)} \quad \forall \text{ communications } A$$

Theorem 6.9.4 *For the above system,* $*[(S_1)^{k_1}] \parallel *[(S_2)^{k_2}]$ *has all its communications well-balanced.*

Proof: See the proof in theorem 6.9.3. □

Corollary 6.9.1 *For any feasible system with the minimal form SLHE proceses* S_1, S_2: $*[S_1] \parallel *[S_2]$, *there exists a constant* k *such that the following relation holds for any communication* X *between the two processes:*

$$\frac{n(X)}{m(X)} = k$$

where $n(X)$ is the number of communications X in S_1, and $m(X)$ is the number of communications X in S_2.

Proof: It's trivial from the previous theorem that $k = \frac{k_2}{k_1}$. And $*[(S_1)^{k_1}] \parallel *[(S_2)^{k_2}]$ has all communications well-balanced, $\gcd(k_1, k_2) = 1$. □

Corollary 6.9.2 (consistency) *For a system: $*[S_1] \parallel \ldots * [S_n]$, if $*[(S_1)^{k_1}] \parallel \ldots \parallel *[(S_n)^{k_n}]$ has all the communications well-balanced then for any communication X between any two processes S_i, S_j we have:*

$$\frac{n(X)}{m(X)} = \frac{k_j}{k_i}$$

Balancing Systems with a Finite Number of Processes

Now I will study the problem of balancing the communications in a given system with n SLHE processes ($n \geq 2$): $*[S_1] \parallel \ldots \parallel *[S_n]$ (where all S_i are in their minimal form).

In a previous section 6.2, I introduced the concept of a configuration graph. Here all systems have connected configuration graphs. It is a reasonable assumption because unconnected subparts are physically separated and there exists no deadlock across unconnected blocks.

Because of the corollary on consistency, if two processes communicate to each other via more than one channel and if communication on one channel is well-balanced, then communication on all the channels between the two are well-balanced.

Lemma 6.9.3 *Given the configuration graph of the system, pick one channel between every two nodes for which there are multi-channels between them, call this the single-edged configuration graph. All the communications in the system are well-balanced if and only if all the communications on the selected channels (in the single-edged configuration graph) are well-balanced.*

Proof: The only if part is more than trivial. I will prove the "if" part: Arbitrarily pick a communication channel X connecting process node P_i and P_j. If the channel X is selected in the single-edged configuration graph then by assumption it is well-balanced and we are done. If X is not selected then there must be another channel Y between the two nodes which is selected

and by assumption communication on Y is well-balanced. According to the corollary of consistency: communication on channel X is well-balanced if communication on Y is well-balanced. Therefore X is well-balanced too. □

However, we can do even better:

Lemma 6.9.4 *Given the (connected) single-edged configuration graph of a system, pick a spanning tree of the graph. All the communications in the system are well-balanced if and only if all the communications in the spanning tree are well-balanced.*

Proof: If two nodes P_1 and P_n are connected by some channel X in the original single-edged configuration graph, then there exists a unique path $(P_1, P_2, \ldots, P_n)$ in the spanning tree connecting the two nodes via channels $C_{12}, C_{23}, \ldots C_{(n-1)n}$. Denote the number of communications on a channel X in S_i of a process $P_1 \equiv *[S_i]$ as $N_{P_i}(X)$, if the system $*[(S_1)^{k_1}] \parallel \ldots \parallel *[(S_n)^{k_n}]$ has all its communications well-balanced on the communications in the spanning tree such as channel $C_{12}, \ldots C_{(n-1)n}$, then it is well-balanced on X because

$$\begin{aligned} \frac{N_{P_1}(X)}{N_{P_n}(X)} &= \frac{N_{P_n}(C_{(n-1)n})}{N_{P_{n-1}}(C_{(n-1)n})} \cdot \frac{N_{P_{n-1}}(C_{(n-2)(n-1)})}{N_{P_{n-2}}(C_{(n-1)(n-2)})} \cdots \frac{N_{P_2}(C_{12})}{N_{P_1}(C_{12})} \qquad (*) \\ &= \frac{k_n}{k_{n-1}} \cdot \frac{k_{n-1}}{k_{n-2}} \cdots \frac{k_2}{k_1} \\ &= \frac{k_n}{k_1} \end{aligned}$$

Here I assumed the (*) relation holds which I will prove in the corollary.

Therefore, if all the communications in the spanning tree of the single-edged configuration graph are well-balanced then all the communications in the single-edged configuration graph are well-balanced. By the previous Lemma 6.9.4 all the communications in the system are well-balanced. □

Corollary 6.9.5 (balancing condition) *A system of SLHE processes $*[S_1] \parallel \ldots \parallel *[S_n]$ can have all its communications balanced if and only if the communications satisfy the following relations:*

1. *There is a fixed ratio $\frac{k_j}{k_i}$ between two processes P_i, P_j which communicate with each other, such that for any communication X between the two processes:*

$$\frac{N_{P_i}(X)}{N_{P_j}(X)} = \frac{k_j}{k_i}$$

Because of this we can denote this ratio $\frac{k_j}{k_i}$ *as* $\frac{N_{P_i}}{N_{P_j}}$.

2. *For* n *communicating processes* $P_1, \ldots, P_n$, *if* P_i *communicates with* P_{i+1} $(1 \leq i \leq n-1)$, *and* P_1 *communicates with* P_n *then the following must hold:*

$$\frac{N_{P_1}}{N_{P_n}} = \prod_{i=1}^{n-1} \frac{N_{P_i}}{N_{P_{i+1}}}$$

Proof: The first condition is a rewording of the consistency condition.

The second condition can be proved as follows: If the system can be balanced to the system $*[(S_1)^{k_1} \parallel \ldots \parallel *[(S_n)^{k_n}]$ then

$$\begin{aligned} \frac{N_{P_1}}{N_{P_n}} &= \frac{k_n}{k_1} \\ &= \frac{k_n}{k_{n-1}} \cdot \frac{k_{n-1}}{k_{n-2}} \cdots \frac{k_2}{k_1} \\ &= \frac{N_{P_n}}{N_{P_{n-1}}} \cdot \frac{N_{P_{n-1}}}{N_{P_{n-2}}} \cdots \frac{N_{P_2}}{N_{P_1}} \end{aligned}$$

□

Theorem 6.9.5 *For a given system* $*[S_1] \parallel \ldots \parallel *[S_n]$ *which satisfies the balancing condition, there exists a unique* n*-tuple* $(k_1, k_2, \ldots, k_n)$, *such that*

- $\gcd(k_1, \ldots, k_n) = 1$ *and*
- *if process* $P_i = *[S_i]$ *communicates with process* $P_j = *[S_j]$, *then*

$$\frac{N_{P_i}}{N_{P_j}} = \frac{k_j}{k_i}.$$

System $*[(S_1)^{k_1}] \parallel \ldots \parallel *[(S_n)^{k_n}]$ *has all communications well-balanced.*

I will give a constructive proof by giving the following algorithm for finding the n-tuple $(k_1, \ldots, k_n)$.

Span-Tree Unrolling Algorithm:

Suppose we have built the spanning tree of the single-edged configuration graph of the system, each node V_i corresponds to a process $P_i \equiv *[S_i]$, and

each edge (V_i, V_j) corresponds to a channel C_{ij} between processes P_i and P_j.

Def: Given a process $*[S_1]$, it is *unwrapped k* times if it is transformed to $*[(S_1)^k]$. The number k is called the *unwrapping number.*

The data structure of a node V_i contains the following fields:

- $V_i.N(C_{ij})$: the number of communications on the channel C_{ij} in S_i of process $P_i = *[S_i]$.
- $V_i.uwroot$: unwrapping number of V_i as the root of its subtree if the subtree is being balanced.
- $V_i.uwchild$: unwrapping number of the balanced subtree rooted at V_i if the subtree rooted at the parent of V_i is being balanced.

Now I will present the algorithm:

1. Initialize $V_i.N(C_{ij}) = N_{V_i}(C_{ij})$, $V_i.uwroot = 1$, $V_i.uwchild = 1$ for all the process nodes V_i.
2. Balance all the subtrees rooted at the parents of the leaf nodes (subtrees of height 1) in the following way:
   - For the root node V_0 with leaf nodes $V_1, \ldots, V_n$, let $w_i = \frac{lcm(V_0.N(C_{0i}), V_i.N(C_{0i}))}{V_0.N(C_{0i})}$ for $i = 1, \ldots, n$.

     $$V_0.uwroot = lcm(w_1, \ldots, w_n)$$

   - For a leaf node V_i $(1 \leq i \leq n)$,

     $$V_i.uwchild = \frac{V_0.uwroot \cdot V_0.N(C_{0i})}{V_i.N(C_{0i})}$$

3. Now we can build a new tree with all its leaf nodes being the balanced subtrees of height 1 in the original spanning tree graph. The new leaf node $\hat{V}_0$ which corresponds to the subtree rooted at V_0 has the following data in its fields:

   $$\hat{V}_0.N(C_{0k}) = V_0.uwroot \times V_0.N(C_{0k}),\ \hat{V}_0.uwroot = 1,\ \hat{V}_0.uwchild = 1$$

   Then we will repeat the algorithm in step 2. And finally we have

   $$V_0.uwchild = \hat{V}_0.uwchild$$

4. Similar to step 3, we build a new tree with all its leaf nodes being the balanced subtrees of height 2 of the original spanning tree. Repeat the algorithm in step 3.

5. Repeat step 4 until the new tree has all its leaf nodes being the balanced subtrees rooted at the children of the root of the original subtree.

6. Now we calculated the $V_i.uwroot$ and $V_i.uwchild$ for each node V_i in the spanning tree. In particular, the root node R has $R.uwchild = 1$, and a leaf node L has $L.uwroot = 1$. Then the unwrapping number for a process V_i in the final balanced system is

$$k_i = (\prod_{V_j \in path(root, V_i)} V_j.uwchild) V_i.uwroot$$

And the above algorithm gives the n-tuple $(k_1, \ldots, k_n)$ such that $*[(S_1)^{k_1}] \parallel \ldots \parallel *[(L_n)^{k_n}]$ has all its communication well-balanced.

Proof of the Span-Tree Unrolling Algorithm
We want to show the k_i given by the algorithm satisfy the following property: For any edge X in the spanning tree connecting process $P_i = *[S_i]$ and process $P_j = *[S_j]$,

$$\frac{k_j}{k_i} = \frac{N_{P_i}(X)}{N_{P_j}(X)}$$

and also $\gcd(k_1, \ldots, k_n) = 1$.

Proof: Without loss of generality suppose that node V_i is the parent of node V_j, so $path(root, V_j) = path(root, V_i) +\!+ (V_i, V_j)$. From the span-tree unrolling algorithm, we have:

$$\begin{aligned} k_i &= (\prod_{V_l \in path(root, V_i)} V_l.uwchild) V_i.uwroot \\ k_j &= (\prod_{V_l \in path(root, V_j)} V_l.uwchild) V_j.uwroot \end{aligned}$$

Therefore,

$$\frac{k_i}{k_j} = \frac{V_i.uwroot}{V_j.uwchild \times V_j.uwroot}$$

But from step 3 of the algorithm, it is easy to show that:

$$V_i.uwroot \times V_i.N(X) = V_j.uwchild \times (V_j.uwroot\ V_j.N(X)).$$

Therefore

$$\frac{k_i}{k_j} = \frac{V_j.N(X)}{V_i.N(X)} = \frac{N_{V_j}(X)}{N_{V_i}(X)}.$$

Done.

Last, we will show that $\gcd(k_1, \ldots, k_n) = 1$. Let k_r be the unwrapping number of the root, and k_{s_i} be the unwrapping number of a child V_{s_i} of the root V_r. Then

$$\begin{aligned} k_r &= V_r.uwroot \\ k_{s_i} &= V_{s_i}.uwchild \times V_{s_i}.uwroot \quad \forall \text{ childen } V_{s_i} \end{aligned}$$

From step 2 in the algorithm, we have $\gcd(k_r, k_{s_1}, \ldots, k_{s_m}) = 1$, but we also have $\{k_r, k_{s_1}, \ldots, k_{s_m}\} \subset \{k_1, \ldots, k_n\}$, therefore $\gcd(k_1, \ldots, k_n) = 1$. □

6.10 Conclusion

In this Chapter, I developed the theory of detecting deadlocks in a closed system with only SLHE processes. The system is deadlock-free if its corresponding communication dependency graph is acyclic. However, because the CDG is infinite it makes the detection impractical. In order to make the detection possible and efficient, I studied a special type of system: repetitive systems, where the detection of deadlocks is simple and only deals with a finite graph. Finally, I present a method to transform the detection of deadlocks in the original system to an equivalent detection problem in a corresponding repetitive system. The complexity of the transformation from the original system to the corresponding repetitive system is linear with respect to the number of processes in the system, and the detection of cycles in a finite graph is not very complicated.

Chapter 7

Deadlock Detection on Systems with GC Processes

7.1 Introduction

In this Chapter, I will study the detection of deadlock in general systems. The generalization goes in the following two directions:

- The system has not only SLHE processes, but also processes with deterministic guarded commands [1], which I will refer to as GC processes.
- The environment will be modeled not only by SLHE processes, but also by GC processes. Various environment scenarios will be taken into account.

By allowing guarded commands, choices are introduced into the system. This generalization includes most systems we design, such as the asynchronous microprocessor [2] etc.

The major difficulty in the liveness analysis of the general system is that the system may have many computations. Unlike the SLHE system that we studied in Chapter 6, we need to calculate the computation (or the successor relations) before building the corresponding communication dependency

[1] No arbitration is allowed. The class of systems with no arbitration is big, it includes the first Caltech asynchronous microprocessor.

[2] The asynchronous microprocessor could be modeled as a big GC process with initialization (the original CSP). By providing an SLHE process for the environment we can get a closed system with only SLHE processes and GC processes.

graph and then use the theory developed in Chapter 6. Also the computations may not belong to the set of SLHE computations.

Def: A computation is called a *SLHE computation* if it can be produced by an SLHE system, a closed system of straight-line handshaking expansion processes. [3]

I will show that there is a unique computation for each general (closed) system with SLHE processes and GC processes. Furthermore, the computation is an SLHE computation.

7.2 Computational Equivalency

Def: Two SLHE systems are *computationally equivalent* if they have the same computation. More generally, two systems are computationally equivalent if all their computations are the same.

For example, system A:

$$\begin{aligned} P_1 &\equiv *[D; E; C; F] \\ P_2 &\equiv *[D; E] \\ P_3 &\equiv *[C; F] \end{aligned}$$

and system B:

$$\begin{aligned} P_1 &\equiv *[D; E; C; F] \\ P_2 &\equiv *[D; C] \\ P_3 &\equiv *[E; F] \end{aligned}$$

are computationally equivalent in spite of their syntactical difference. Therefore an SLHE computation can have many SLHE systems that generate it.

A general system with GC processes is computationally equivalent to an SLHE system if we can turn all the GC processes in the system into SLHE processes while the computational equivalency of the system is still

[3] In Chapter 6 I have shown that each SLHE system has a unique computation.

preserved. There are two simple ways to transform a GC process into an SLHE process(es):

- Turn each guarded command into an SLHE process (state variables might be introduced during the transformation).
- Replace the GC process with an SLHE process consisting of guarded commands sequenced in the order of execution.

Theoretically, both transformations can be applied to any GC process. The general GC process can be divided into two groups: one with mutually exclusive (or *mutex*) guarded commands, and the other with non-mutex guarded commands. For GC processes with mutex guarded commands, the first transformation has the advantage over the second one; for GC processes with non-mutex guarded commands, the second transformation has advantage over the first transformation.

7.3 The Computation of a General System

Def: A *computation of a GC process* is a sequencing of its guarded commands according to the excution order in some computation of the system.

For example, for the GC process

$$*[\bar{A} \longrightarrow S_1; A \parallel \bar{B} \longrightarrow S_2; B]$$

one of its computations could be:

$$[[A]; S_1; A; [B]; S_2; B; [A]; \ldots]$$

In particular, the *computation of an SLHE process* $*[S]$ is $[S; S; S; \cdots]$. A computation of the system is a parallel composition of all the computations of its processes.

Theorem 7.3.1 *For a closed system with SLHE processes and (deterministic) GC processes there is a unique computation.*

Proof: Proof by contradiction. Suppose that the system has more than one computation, pick any two: c_1 and c_2. The two computations must differ at the computation(s) of some GC process(es), that is, in c_1 the computation for the GC process P_k is $[G_1; G_2; G_3; \ldots]$ while in c_2 the computation for the

process P_k is $[G_1; G_2; G_4; \ldots]$, where the G_i are guarded commands[4] in P_k. I refer to the point where the two computations differ, i.e. the semicolon after G_2, as the *deviation point* between computations of the GC process P_k. Now let's construct execution traces of the two distinct computations. The two compuations c_1 and c_2 share the same trace until one of the guards at a deviation point of some GC process becomes true, for example, the guard for G_3 becomes true. According to c_1 we can construct an execution trace t_1 (for c_1) such that the corresponding guarded command G_3 is taken; but according to c_2, since G_3 is not taken, we have an execution trace t_2 (for c_2) such that the guard of G_3 remains true but untaken until the guard of G_4 becomes true, whereat both guards are true. This contradicts the condition that the guards of the GC process are mutually exclusive in any execution. □

Def: A *period of the guarded command G in the computation t of the GC process Q* is an integer p such that in the computation t all the $(k + pi)$th guarded commands are G. Here k is an integer and $i = 0, 1, 2, \ldots$.

For example, in the computation $[G_1; G_2; G_1; G_2; \ldots]$ where the sequence $G_1; G_2$ repeats itself forever, the period of both guarded commands G_1 and G_2 is 2.

Lemma 7.3.1 *Given any general system with SLHE processes and (deterministic) GC processes, there exists at least one GC process P such that all its guarded commands have a common period in the computation of P.*

Proof: In fact we will prove that all GC processes P have the asserted property. Proof by contradiction. Pick any GC process P. Suppose there exists no common period for its guarded commands in the computation of the system. Since the system is data-independent, i.e. the state of the system is determined by the state of the variables in each process, and since there are only finitely many states in each process, we can pick any execution trace t (with infinite length) and find two identical states (of the system), σ_A and σ_B. Then we can construct a new execution t^* from t as follows: t and t^* share the same trace until state σ_A happens for the first time, then t^* will repeat the part of the trace which transforms the state σ_A to σ_B.
This new execution trace t^* should be produced by the unique computation of the general system (theorem 7.3.1) and this implies that for the GC process P

[4]In this Chapter, I assume that all guarded commands are sequences of waits and transitions.

the execution order of its guarded commands must be repetitive, equivalently all the guarded commands have a common period. Contradiction. □

Corollary 7.3.2 *If there exists a common period for all the guarded commands in the computation of the GC process P, then we can turn P into a computationally equivalent SLHE process Q.*

For example, the GC process with computation

$$[G_1; G_2; G_1; G_2; \ldots]$$

can be turned into the SLHE process $*[G_1; G_2]$.

Theorem 7.3.2 *For a closed system with SLHE processes and GC processes the unique computation is an SLHE computation.*

Proof: I've already shown in theorem 7.3.1 that the system has a unique computation; now I will show that the compuation can be modeled by an SLHE system by showing that all the GC processes can be turned into SLHE processes (with some initialization procedure). According to the previous corollary, for any general system with SLHE and GC processes there exists at least one GC process that can be turned into an SLHE process. Therefore we get a computationally equivalent system after transforming a GC process into an SLHE process. Using the corollary again, we can find another GC process to turn into an SLHE process. Continuing in this manner we will finally derive a system with only SLHE processes since the number of GC processes in the original system is finite. And the SLHE system is computationally equivalent to the general system, therefore the computation for a closed system with SLHE processes and GC processes is an SLHE computation. □

Remark: There are many computationally equivalent SLHE systems that generate the same SLHE computation.
(end of Remark)

Here I established the theory that the general system of SLHE processes and GC processes has a unique SLHE computation and thus can be turned into some computationally equivalent SLHE system on which the deadlock detection analysis developed in Chapter 6 can be applied. In the remainder of the chapter I will study methods that transform the general system into a computationally equivalent SLHE system.

7.4 Mutually Exculsive Guarded Commands

Def: The guards of guarded commands are *mutually exclusive* if for any execution there is at most one guard evaluated to **true**. The guarded commands are *mutually exclusive* if in any computation during the execution of a guarded command the guards of the other guarded commands of the same process remain **false**.

Theorem 7.4.1 *A process with guarded commands*

$$
\begin{array}{ll}
*[[& G_1 \longrightarrow S_1 \\
 & \vdots \\
 & G_n \longrightarrow S_n \\
]] &
\end{array}
$$

can be turn into an equivalent set of straight-line programs

$$*[[G_1]; S_1] \parallel \ldots \parallel *[[G_n]; S_n]$$

if all the guarded commands are mutually exclusive.

Proof: It is obvious that the original system is a refinement of the transformed system because there are fewer possible execution traces in the original system. Therefore we only need to show is that the transformed system will not have more execution traces than the original system. Because all the guarded commands are mutually exclusive, there is at most one of the n straight-line programs executed at any time. This is computationally equivalent to a selection of the corresponding guarded command in the original system. Therefore the two systems will have the same set of executions, and they are computationally equivalent. □

7.4.1 Splitting

Def: *Splitting* is a transformation that turns a GC process with n guarded commands

$$
\begin{array}{ll}
*[[& G_1 \longrightarrow S_1
\end{array}
$$

$$
\begin{array}{l}
\quad\vdots \\
\quad G_n \longrightarrow S_n \\
]]
\end{array}
$$

into the parallel composition of n straight-line processes

$$*[[G_1]; S_1] \parallel \ldots \parallel *[[G_n]; S_n].$$

Theorem 7.4.2 *If the given closed system consists of only SLHE processes and of GC processes with mutually exclusive guarded commands and there is no communication shared between guarded commands of the same GC process, then we can transform the system into a computationally equivalent SLHE system by applying the splitting transformation to all the GC processes.*

Proof: We have already proved that splitting transforms the general system into a compuationally equivalent system with only SLHE processes in theorem 7.4.1. Since the guarded commands do not share any communication, each channel still connects two processes after splitting, and we have an SLHE system after the transformation. □

7.4.2 Renaming

However, if splitting is applied to a closed system where all the GC processes have mutually exclusive guarded commands but some have communications shared between guarded commands, then the transformed system with SLHE processes might not be an SLHE system because the transformation will split these shared communications into different processes and violate the rule that ports can't be shared between processes (or syntactically, the same type of comunications will appear in exactly two processes). And we can not apply the deadlock detection algorithm presented in the Chapter 6 on the transformed system.

For example, consider

$$P_1 \equiv *[A; B]$$

$$
\begin{aligned}
P_2 &\equiv *[E; F; D; F] \\
P_3 &\equiv *[[\ \bar{A} \longrightarrow E; F; A \\
&\quad\ [\!]\ \bar{B} \longrightarrow D; F; B \\
&\quad\ ]]
\end{aligned}
$$

Since all the guarded commands of P_3 are mutually exclusive, we transform it into the following system with SLHE processes:

$$
\begin{aligned}
P_1 &\equiv *[A; B] \\
P_2 &\equiv *[E; F; D; F] \\
P_{3a} &\equiv *[[\bar{A}]; E; F; A] \\
P_{3b} &\equiv *[[\bar{B}]; D; F; B]
\end{aligned}
$$

Notice that F is shared between the processes $P_2,\ P_{3a},\ P_{3b}$. The solution to the above problem is simple. The system has a unique SLHE computation and in this case it is easy to find the computation, which can be produced by the following SLHE system:

$$
\begin{aligned}
P_1 &\equiv *[A; B] \\
P_2 &\equiv *[E; F; D; F] \\
P_3 &\equiv *[[\bar{A}]; E; F; A; [\bar{B}]; D; F; B]
\end{aligned}
$$

We will rename the F communication in the first guarded command as F_1 and the F communication in the second guarded command as F_2. Therefore we have the renamed SLHE system:

$$
\begin{aligned}
P_1 &\equiv *[A; B] \\
P_2 &\equiv *[E; F_1; D; F_2] \\
P_3 &\equiv *[[\bar{A}]; E; F_1; A; [\bar{B}]; D; F_2; B]
\end{aligned}
$$

Combine the splitting and renaming procedure to obtain the following SLHE system:

$$
\begin{aligned}
P_1 &\equiv *[A; B] \\
P_2 &\equiv *[E; F_1; D; F_2] \\
P_{3a} &\equiv *[[\bar{A}]; E; F_1; A] \\
P_{3b} &\equiv *[[\bar{B}]; D; F_2; B]
\end{aligned}
$$

And we can now perform the algorithm in Chapter 6 to detect the deadlock.

Theorem 7.4.3 *A general system with SLHE processes and GC processes with mutually exclusive guarded commands can be transformed into a computationally equivalent SLHE system by splitting and renaming.*

Proof: We have proved that the general system has a unique SLHE computation (theorem 7.3.2). Therefore we can obtain a computationally equivalent SLHE system Sys_A which is the parallel composition of the SLHE processes in the original system and the SLHE processes transformed from the GC processes. Now I will derive another computationally equivalent SLHE system Sys_B by splitting the guarded commands in a GC process and by renaming the shared communications as follows:

- First, by splitting, transform all the GC processes into SLHE processes. Rename all the communications shared between guarded commands. Include all the SLHE processes transformed from the GC processes in Sys_B. Denote the set of communications that are renamed as R.
- Then in Sys_A rename the communications in the set R for each process.
- Include in Sys_B all the SLHE processes from the renamed Sys_A which are also SLHE processes in the original system.

□

7.5 Non-mutually Exclusive Guarded Commands

If a GC process has non-mutually exclusive guarded commands, then by splitting the process into several SLHE processes each of which contains one guarded command we will introduce interference into the parallel execution of the transformed SLHE processes.

One solution to this problem is to strengthen the guards of the original guarded commands to make them mutually exclusive. This can be done if we allow state variables to be added. However by splitting, we will make the

state variables shared between processes which violates one taboo assumed before in this thesis: no shared variables between processes. Shared variables can be handled in the same manner as the communications which in essence are operations on shared communication variables except we have to watch out for vacuous waits and transitions in the liveness analysis. For example, consider the following system:

$$\begin{aligned} P_1 &\equiv *[A; E; B; D] \\ P_2 &\equiv *[F] \\ P_3 &\equiv *[[\ \bar{A} \longrightarrow A; E; F \\ &\quad \| \ \bar{B} \longrightarrow B; D; F \\ &\quad]] \end{aligned}$$

Notice that P_3 doesn't have mutex guarded commands. By splitting we obtained:

$$\begin{aligned} P_1 &\equiv *[A; E; B; D] \\ P_2 &\equiv *[F] \\ P_{3a} &\equiv *[[\bar{A}]; A; E; F] \\ P_{3b} &\equiv *[[\bar{B}]; B; D; F] \end{aligned}$$

We will have interference at the F communications. Therefore, we must strengthen the guards of the original guarded commands to make them mutually exclusive.

$$\begin{aligned} P_1 &\equiv *[A; E; B; D] \\ P_2 &\equiv *[F] \\ P_3 &\equiv *[[\ \bar{A} \wedge \neg t \longrightarrow s\uparrow; A; E; F; s\downarrow \\ &\quad \| \ \bar{B} \wedge \neg s \longrightarrow t\uparrow; B; D; F; t\downarrow \\ &\quad]] \end{aligned}$$

Then we can apply the splitting transformation and get:

$$\begin{aligned} P_{3a} &\equiv *[[\bar{A} \wedge \neg t]; s\uparrow; A; E; F; s\downarrow] \\ P_{3b} &\equiv *[[\bar{B} \wedge \neg s]; t\uparrow; B; D; F; t\downarrow] \end{aligned}$$

Finally we obtain the SLHE system with a shared variable s as follows:

$$\begin{aligned} P_1 &\equiv *[A; E; B; D] \\ P_2 &\equiv *[F_1; F_2] \\ P_{3a} &\equiv *[[\bar{A} \wedge \neg t]; s\uparrow; A; E; F_1; s\downarrow] \\ P_{3b} &\equiv *[[\bar{B} \wedge \neg s]; t\uparrow; B; D; F_2; t\downarrow] \end{aligned}$$

The above system is computationally equivalent to the original system.

However in the above program, $[\neg t]$ could be an ineffective wait, unlike the waits on the communication variables, therefore we have to keep track of which effective firing of t caused the wait $[\neg t]$ to be true in the liveness analysis.

So in this case the SLHE system that is obtained from the SLHE computation is more favorable:

$$\begin{aligned} P_1 &\equiv *[A; E; B; D] \\ P_2 &\equiv *[F_1; F_2] \\ P_3 &\equiv *[[\bar{A}]; A; E; F_1; [\bar{B}]; B; D; F_2] \end{aligned}$$

7.6 Summary of the Transformation Methods

Theoretically, there exists a unique SLHE computation for any closed system with only SLHE processes and (deterministic) GC processes and we can turn the system into a computationally equivalent SLHE system and then apply the deadlock detection algorithm studied in Chapter 6.

If for each GC process its guarded commands share no communication and are mutually exclusive, then we can transform the system into an SLHE system by applying splitting to all the GC processes.

If some guarded commands are not mutually exclusive, we can introduce state variables, "split" the GC processes, and find the vacuous waits on the state variables. If there are communications shared between some guarded commands, then we have to rename the shared communications.

7.7 System with Data Dependency

For general systems with some data dependency they might not have SLHE computations. For example:

$$\begin{aligned} P_1 &\equiv *[A!y; y = f(y)] \\ P_2 &\equiv *[B] \\ P_3 &\equiv *[[\ x \longrightarrow A?x; B \\ &\quad [\!] \neg x \longrightarrow B; A?x \\ &\quad]] \end{aligned}$$

Variable x could get such an irregular sequence of values as $1, 0, 1, 1, 0, 1, 1, 1, 0 \ldots$ which gives P_3 the following computation:

$$[A; B; B; A; A; B; A; B; B; A; A; B; A; B; A; B; B; A; \cdots]$$

Therefore it is impossible to transform the above system into an SLHE system and we may not perform the liveness analysis that we have developed.

Chapter 8

Generation of Deadlock-free Reshuffling

8.1 Introduction

The question being addressed in this chapter is the generation of deadlock-free reshufflings. The nature of the problem has the following aspects which differ from the detection of DF reshufflings studied in the previous chapter:

- The detection method treats all systems and reshufflings in a uniform way and could be very inefficient when applied to some simple reshuffling of a large system with many processes or with a complicated communication pattern.
  The generation method, however, will use heuristics to generate DF reshufflings with respect to distinct systems and environment behavior; correctness is guaranteed by generation.

- The detection method works for fixed environment scenarios; while the generation method would allow more general environment behavior: for a complicated environment, we can give several environment scenarios and find the sets of DF reshufflings, respectively, then the intersection of these sets of DF reshufflings will be deadlock-free for the union of the environment scenarios.

- The generation method could provide a set of deadlock-free reshufflings from which we can choose efficient ones.

In order to find heuristics for the generation of DF reshufflings, I am going to study reshuffling from a brand-new angle: I will study the algebraic

properties of reshuffling. The main idea is to treat reshuffling as a group action on a set of handshaking expansions, and deadlock-free reshuffling as a subgroup of reshufflings which *acts* on the set of deadlock-free handshaking expansions. By finding the generators of the subgroup of DF reshuffings, we can construct the whole subgroup using the algebraic properties of a group.

So the main difficulty here is to build a simple but exact mathematical model for reshuffling and to study its algebraic properties. In the following section, I will first review some simple mathematical concepts on group actions etc., then I will establish the group action model for reshuffling.

8.2 Reshuffling as a Group Action

Given a closed system consisting of SLHE (straight-line handshaking expansion) processes and GC (guarded command) processes, it has a unique computation, which can be written as a closed system with only SLHE processes. Therefore I only consider the reshufflings on an SLHE system.

Mathematically, reshuffling on a closed SLHE system can be viewed as an action that transforms the closed SLHE system to another closed SLHE system.

Every closed SLHE system has a distinctive CDG (communication dependency graph). The subset of the closed SLHE systems with acyclic CDGs is called the set of *deadlock-free closed SLHE systems* (abbrevaited as DF-SLHE systems). A *deadlock-free reshuffling* (abbreviated as DF reshuffling) is a reshuffling that transforms all DF-SLHE systems to other DF-SLHE systems. Our job is to identify the subset of DF reshufflings inside the set of reshufflings. Furthermore, I will show the set of DF reshufflings has a group structure, therefore we can generate new DF reshufflings from the given DF reshufflings.

8.2.1 Group action

It is an important and recurring idea in mathematics that when one object *acts* on another object then a lot of information, especially the algebraic properties, can be obtained on both. The case of group actions is important in mathematics. An example of a group action is permutation. But first, I will give the definition of a group action [3].

Def: A *(right) group action* of a group G on a set A is a map from $A \times G$ to A(written as $a \cdot g$, for all $g \in G$ and $a \in A$) satisfying the following properties:

(1) $(a \cdot g_1) \cdot g_2 = a \cdot (g_1 \circ g_2)$, for all $g_1, g_2 \in G, a \in A$, and

(2) $a \cdot 1 = a$, for all $a \in A$.

We say G is a group acting on a set A.

Some Remarks:
We use the less popular notion of a right group action (instead of the left group action) because when $g_1 \circ g_2$ acts on a we have first g_1 acting on a then g_2 acting on $a \cdot g_1$. We use the right group action to avoid confusion about order when performing composition. $(a \cdot g_1) \cdot g_2$ can be read as a acted on by g_1 then by g_2, we can even remove the bracket here.
(end of remark)

Let the group G act on the set A. For each fixed $g \in G$ we get a map σ_g defined by

$$\sigma_g : A \mapsto A \text{ such that } \sigma_g(a) = a \cdot g$$

It has been shown that:

1. for each fixed $g \in G$, σ_g is a *permutation* of A, and
2. the map from G to S_A(the permutation group) defined by $g \mapsto \sigma_g$ is a group homomorphism.

8.2.2 Re-definition of Reshuffling

Given a directly-mapped SLHE system P, i.e. the original handshaking expansion translated from the CSP, there are finite number of reshuffled SLHE systems which can be obtained from the original SLHE system, denoted as $\mathbf{res}(P)$.

According to the original definition given by Dr. Martin, reshuffling is an action of postponing the resetting part of a communication in a handshaking program. However, this creates some problems in our context: let Q be a reshuffled system of P, then $\mathbf{res}(Q) \subset \mathbf{res}(P)$ which makes a group action difficult to define. The cause of the problem is that in the original definition reshuffling is defined as a postponing action, whereas all that we really should demand is to rearrange the resetting part of a communication so that the

communication protocol is preserved. So here is my modified definition of reshuffling:

Def: A *directly-mapped SLHE system* is a closed system with only SLHE processes such that every communication in any SLHE process is expanded by their corresponding handshaking protocol; in other words, the resetting part D_x immediately follows the upgoing part U_x as $U_x; D_x$ for a communication X.
Given an SLHE system Q, $\mathbf{res}(Q)$ includes the following SLHE systems:

- $Q \in \mathbf{res}(Q)$.
- If Q is a directly-mapped SLHE system, then $\mathbf{res}(Q)$ includes all the SLHE systems obtained by postponing the resetting part of some communications in the SLHE processes.
- If Q is a reshuffled form of the directly-mapped SLHE system P, then $P \in \mathbf{res}(Q)$.
- If $S \in \mathbf{res}(Q)$, then $\mathbf{res}(S) \subseteq \mathbf{res}(Q)$.

Corollary 8.2.1 *Let Q be a reshuffled system of the directly-mapped SLHE system P, then* $\mathbf{res}(Q) = \mathbf{res}(P)$.

Proof: Easy by the definition of **res**. □

Note: According to the above definition, not only is $*[U_x; U_y; D_x; D_y]$ a reshuffling of $*[U_x; D_x; U_y; D_y]$, but also vice versa.
(end of Note)

Def: Given an SLHE system P, *a reshuffling of P* is a bijection on $\mathbf{res}(P)$.

8.2.3 Reshuffling as a Group Action

Now given an SLHE system P, the set of reshufflings R contains all the bijections from $\mathbf{res}(P) \mapsto \mathbf{res}(P)$. I will show that R forms a group, and that the group R acts on the set $\mathbf{res}(P)$ by mapping from $\mathbf{res}(P) \times R$ to $\mathbf{res}(P)$.

Lemma 8.2.2 *R satisfies the following properties of a group action:*

(0) $p \cdot r \in \mathbf{res}(P)$, *for all* $p \in \mathbf{res}(P), r \in R$.

(1) Let $p \cdot (r_1 \circ r_2) = (p \cdot r_1) \cdot r_2$, *for all* $r_1, r_2 \in R$, $p \in \mathbf{res}(P)$, *then* $r_1 \circ r_2 \in R$.

(2) $\exists\ 1 \in R$, *s.t.* $p \cdot 1 = p$, *for all* $p \in \mathbf{res}(P)$.

(3) $\forall r \in R, p_1 \in P, p_2 \in P,\ p_1 \neq p_2 \Rightarrow p_1 \cdot r \neq p_2 \cdot r$.

Proof:

(0) Trivial by the definition of R.

(1) Because r_1 and r_2 are bijections on $\mathbf{res}(P)$, the composition of two bijections $r_1 \circ r_2$ is also a bijection on $\mathbf{res}(P)$.

(2) $\mathbf{1}$ is the identity bijection on $\mathbf{res}(P)$.

(3) This is an alternative way of saying that$r$ is a bijection on $\mathbf{res}(P)$.

□

The identity element $\mathbf{1}$ means no reshuffling. Because $1 \in R$, R is non-empty.

Lemma 8.2.3 $(R, \circ)$ *forms a group.*

Proof: I will show that $(R, \circ)$ satisfies the following properties of a group structure:

(a) $(r_1 \circ r_2) \circ r_3 = r_1 \circ (r_2 \circ r_3)$, for all $r_1, r_2, r_3 \in R$ (Associativity)

proof: For any $p \in \mathbf{res}(P)$, we have:

$$\begin{aligned} p \cdot ((r_1 \circ r_2) \circ r_3) &= (p \cdot (r_1 \circ r_2)) \cdot r_3 \\ &= ((p \cdot r_1) \cdot r_2) \cdot r_3 \end{aligned}$$

$$\begin{aligned} p \cdot (r_1 \circ (r_2 \circ r_3)) &= (p \cdot r_1) \cdot (r_2 \circ r_3) \\ &= ((p \cdot r_1) \cdot r_2) \cdot r_3 \end{aligned}$$

(b) There exists an element **1** in R such that $r \circ 1 = 1 \circ r = r$, for all $r \in R$ (**1** is called *identity* of R).

proof: For any $p \in \mathbf{res}(P)$, we have:

$$\begin{aligned} p \cdot (r \circ 1) &= (p \cdot r) \cdot 1 \\ &= p \cdot r \end{aligned}$$

$$\begin{aligned} p \cdot (1 \circ r) &= (p \cdot 1) \cdot r \\ &= p \cdot r \end{aligned}$$

(c) For each $r \in R$ there is an element r^{-1} of R such that $r \circ r^{-1} = r^{-1} \circ r = 1$.

proof: Because any $r \in R$ is a bijection on P, therefore there exists a unique bijection which is both the left and right inverse of r.

□

8.2.4 Definition of DF reshuffling

Def: A *DF-SLHE system* is a deadlock-free closed system with only SLHE processes. Assume[1] all directly-mapped SLHE systems are DF-SLHE systems.
Given a DF-SLHE system Q, $\mathbf{dfr}(Q)$ is the deadlock-free subset of $\mathbf{res}(Q)$ such that:

- $Q \in \mathbf{dfr}(Q)$.
- If Q is a directly-mapped SLHE system, then $\mathbf{dfr}(Q)$ includes all the deadlock-free SLHE systems in $\mathbf{res}(Q)$.
- If Q is a reshuffled form of the directly-mapped SLHE system P, then $P \in \mathbf{dfr}(Q)$.
- If $S \in \mathbf{dfr}(Q)$, then $\mathbf{dfr}(S) \subseteq \mathbf{dfr}(Q)$.

[1]This is a reasonable assumption because the directly-mapped SLHE systems are derived from a correct (thus deadlock-free) CSP program via semantic-preserving transformations in the Martin synthesis method.

Corollary 8.2.4 *Let Q be a deadlock-free reshuffled system of the directly-mapped system P, then* $\mathbf{dfr}(Q) = \mathbf{dfr}(P)$.

Proof: By the definition of **dfr**. □

Def: Given a DF-SLHE system P, a *restricted deadlock-free reshuffling of* $\mathbf{dfr}(P)$ is a bijection on $\mathbf{dfr}(P)$.

Let's denote the set of restricted deadlock-free reshufflings as $\hat{R}_{df}$. $\hat{R}_{df}$ is not empty because the identity element $1 \in \hat{R}_{df}$.

Def: Given a DF-SLHE system P, a *deadlock-free reshuffling of* $\mathbf{res}(P)$, r, is a bijection on $\mathbf{res}(P)$ such that r restricted to $\mathbf{dfr}(P)$ is a restricted deadlock-free reshuffling of $\mathbf{dfr}(P)$, i.e. $\forall Q \in \mathbf{dfr}(P),\ Q \cdot r \in \mathbf{dfr}(P)$.

Now I will show that the set of DF reshufflings R_{df} forms a subgroup of reshufflings R. The group R_{df} acts on the set $\mathbf{dfr}(P)$.

Lemma 8.2.5 *R_{df} satisfies the following properties of a group action:*

(0) $p \cdot r \in \mathbf{dfr}(P),\ \forall p \in \mathbf{dfr}(P), r \in R_{df}$.

(1) $(p \cdot r_1) \cdot r_2 = p \cdot (r_1 \circ r_2)\ \forall r_1, r_2 \in R_{df},\ p \in \mathbf{dfr}(P)$, *and* $r_1 \circ r_2 \in R_{df}$.

(2) $1 \in R_{df}$ *and* $q \cdot 1 = q,\ \forall q \in Q$.

Proof:

(0) True because restricted to $\mathbf{dfr}(P)$, r is a bijection on $\mathbf{dfr}(P)$.

(1) Since restricted to $\mathbf{dfr}(P)$ both r_1 and r_2 are bijections on $\mathbf{dfr}(P)$, we know that $r_1 \circ r_2$ is also a bijection on $\mathbf{dfr}(P)$ when restricted to $\mathbf{dfr}(P)$. Therefore, $r_1 \circ r_2 \in R_{df}$.

(2) Trivial.

□

Lemma 8.2.6 $(R_{df}, \circ)$ *is a subgroup of* $(R, \circ)$.

Proof: Since R_{df} is a subset of R, I only need to show that $(R_{df}, \circ)$ forms a group:

(a) $(r_1 \circ r_2) \circ r_3 = r_1 \circ (r_2 \circ r_3)$, for all $r_1, r_2, r_3 \in R_{df}$ (Associativity)

proof: Similar to the proof of (a) for R.

(b) There exists an element $\mathbf{1}$ in R_{df} such that $r \circ 1 = 1 \circ r = r$, for all $r \in R_{df}$.

proof: Similar to the proof of (b) for R.

(c) For each $r \in R_{df}$ there is an element r^{-1} of R_{df} such that $r \circ r^{-1} = r^{-1} \circ r = 1$.

proof: Since $r \in R_{df}$ is a bijection on $\mathbf{res}(P)$, it has both left and right inverse r^{-1}. I will show that $r^{-1} \in R_{df}$: since r restricted to $\mathbf{dfr}(P)$ is a bijection on $\mathbf{dfr}(P)$, therefore r^{-1} restricted to $\mathbf{dfr}(P)$ is also a bijection. Therefore $r^{-1} \in R_{df}$.

□

So far we have shown that given P as a DF-SLHE system, R is a group action on $\mathbf{res}(P)$ and R_{df} is a group action on $\mathbf{dfr}(P)$.

8.3 Constructive Definition of Reshuffling

8.3.1 Operational Definition of Reshuffling

In the previous section, I defined reshuffling as bijections on $\mathbf{res}(P)$ for a given directly-mapped SLHE system P. Here I will give a simple operational definition of reshuffling as a subgroup of the bijections on $\mathbf{res}(P)$ as follows:

Def: Let P be a directly-mapped SLHE system and $Q \in \mathbf{res}(P)$, and let r be a reshuffling that reshuffles Q to T. A trivial way to define the reshuffling r explicitly as a bijection: $\mathbf{res}(P) \mapsto \mathbf{res}(P)$ is

$$\forall S \in \mathbf{res}(P),\ S \cdot r = \begin{cases} T & \text{if } S = Q \\ P & \text{if } S = T \text{ and } T \neq P \\ Q & \text{if } S = P \text{ and } P \neq Q \\ S & \text{if } S \neq P,\ Q,\ T \end{cases}$$

Here we understand that as part of the definition $Q \neq T$ unless both are equal to P, in which case we take r to be the identity map.

This definition restricts reshufflings to a subset of bijections on $\mathbf{res}(P)$. From now on a reshuffling takes this restrictive form.

Def: Given reshufflings r_1 and r_2 as bijections defined above, the composition $r_1 \circ r_2$ forms another reshuffling such that

$$\forall S \in \mathbf{res}(P),\ S \cdot (r_1 \circ r_2) = (S \cdot r_1) \cdot r_2$$

It is well-defined because the composition of two bijections is a bijection, furthermore I will show that the composition is a reshuffling as defined above.

Lemma 8.3.1 *Let r_1 be a reshuffling that reshuffles Q to T, and r_2 be a reshuffling that reshuffles T to R where $Q, T, R \in \mathbf{res}(P)$, then by the above definition of r_1 and r_2 as bijections, composition $r_1 \circ r_2$ forms a reshuffling that reshuffles Q to R (by the above definition).*

Proof: By definition, r_1 is a bijection such that

$$\forall S \in \mathbf{res}(P),\ S \cdot r_1 = \begin{cases} T & \text{if } S = Q \\ P & \text{if } S = T \text{ and } T \neq P \\ Q & \text{if } S = P \text{ and } P \neq Q \\ S & \text{if } S \neq P,\ Q,\ T \end{cases}$$

And, r_2 is a bijection such that

$$\forall S \in \mathbf{res}(P),\ S \cdot r_2 = \begin{cases} R & \text{if } S = T \\ P & \text{if } S = R \text{ and } R \neq P \\ T & \text{if } S = P \text{ and } P \neq T \\ S & \text{if } S \neq P,\ T,\ R \end{cases}$$

The composition (for $Q \neq R$) $r_1 \circ r_2$ forms a bijection that does:

$$\forall S \in \mathbf{res}(P),\ S \cdot (r_1 \circ r_2) = \begin{cases} R & \text{if } S = Q \\ P & \text{if } S = R \text{ and } R \neq P \\ Q & \text{if } S = P \text{ and } P \neq Q \\ S & \text{if } S \neq P,\ Q,\ R \end{cases}$$

Here we must take special care of the case where $Q = R$. However, in this case $r_1 \circ r_2 = 1$ as shown in the next lemma. □

Lemma 8.3.2 *Given any reshuffling r on $\mathbf{res}(P)$ which reshuffles Q to T, then the reshuffling r^{-1} from T to Q is the inverse of r such that $r \circ r^{-1} = r^{-1} \circ r = 1$.*

Proof: By definition, r is a bijection such that

$$\forall S \in \mathbf{res}(P),\ S \cdot r = \begin{cases} T & \text{if } S = Q \\ P & \text{if } S = T \text{ and } T \neq P \\ Q & \text{if } S = P \text{ and } P \neq Q \\ S & \text{if } S \neq P,\ Q,\ T \end{cases}$$

And r^{-1} is a bijection such that

$$\forall S \in \mathbf{res}(P),\ S \cdot r^{-1} = \begin{cases} Q & \text{if } S = T \\ P & \text{if } S = Q \text{ and } Q \neq P \\ T & \text{if } S = P \text{ and } P \neq T \\ S & \text{if } S \neq P,\ Q,\ T \end{cases}$$

And by simple calculation we have:

$$\forall S \in \mathbf{res}(P),\ S \cdot (r \circ r^{-1}) = S \cdot (r^{-1} \circ r) = S.$$

□

Denote the set of reshufflings generated by the operational defintion on $\mathbf{res}(P)$ as $\bar{R}$. Then we have,

Lemma 8.3.3 *$(\bar{R}, \circ)$ is a subgroup of $(R, \circ)$.*

Proof: First, $\bar{R}$ is a subset of bijections on $\mathbf{res}(P)$ therefore is a subset of R. Now we only need to show that

- $1 \in \bar{R}$.
- Given $r_1, r_2 \in \bar{R}$, then $r_1 \circ r_2 \in \bar{R}$.
- Given any $r \in \bar{R}$, then $r^{-1} \in \bar{R}$.

It is trivial that 1 is in $\bar{R}$ when we reshuffle P to P. And by lemma 9.3.1 ??, I have shown that the composition of two reshufflings is a reshuffling; by lemma 9.3.2 ?? I have shown that the inverse of a reshuffling is a reshuffling.

□

8.3.2 Operational Definition of DF Reshuffling

Given a DF-SLHE system P, $\hat{R}_{df}$ is the set of restricted bijections on $\mathbf{dfr}(P)$; we can construct the following subset of bijections on $\mathbf{res}(P)$, $\tilde{R}_{df}$ as follows: $\forall \hat{r} \in \hat{R}_{df}$, then $r \in \tilde{R}_{df}$ is defined by

$$\forall S \in \mathbf{res}(P),\ S \cdot r = \begin{cases} S \cdot \hat{r} & \text{if } S \in \mathbf{dfr}(P) \\ S & \text{if } S \notin \mathbf{dfr}(P) \end{cases}$$

Def: $\bar{R}_{df} = \tilde{R}_{df} \bigcap \bar{R}$.

Lemma 8.3.4 $(\bar{R}_{df}, \circ)$ *forms a subgroup of* $(\bar{R}, \circ)$.

Proof: It is easy to check that $\bar{R}_{df} \subseteq \bar{R}$. Now I will show that

- $1 \in \bar{R}_{df}$.
  proof: Trivial.
- Given $r_1, r_2 \in \bar{R}_{df}$, then $r_1 \circ r_2 \in \bar{R}_{df}$.
  proof: Since r_1, r_2 are bijections restricted to $\mathbf{dfr}(P)$, $r_1 \circ r_2$ is also a bijection restricted to $\mathbf{dfr}(P)$, so $r_1 \circ r_1 \in \tilde{R}_{df}$. And $r_1, r_2 \in \bar{R}$, by lemma 7.3.1?? $r_1 \circ r_2 \in \bar{R}$. Therefore $r_1 \circ r_2 \in \bar{R}_{df}$.
- Given any $r \in \bar{R}_{df}$, then $r^{-1} \in \bar{R}_{df}$.
  proof: Since r is a bijection restricted to $\mathbf{dfr}(P)$, r^{-1} is also a bijection restricted to $\mathbf{dfr}(P)$, therefore $r^{-1} \in \tilde{R}_{df}$. Also $r \in \bar{R}$ gives $r^{-1} \in \bar{R}$. Therefore $r^{-1} \in \bar{R}_{df}$.

□

8.4 Elements in the Group $\bar{R}_{df}$

In the previous sections, I have shown that the set of DF reshufflings forms a group $\bar{R}_{df}$. Therefore if a reshuffling r is the composition of some elements in $\bar{R}_{df}$ then $r \in \bar{R}_{df}$. Ideally, if we can find all the generators of the group $\bar{R}_{df}$ then we can generate all the deadlock-free reshufflings for the given system P. However, this turns out to be very difficult in practice:

- First, to decide whether a reshuffling is in $\bar{R}_{df}$ usually requires a lot of information on the given system P. There are few elements in $\bar{R}_{df}$ that are system-independent, i.e. the *type* of reshuffling is deadlock-free for any system such as the identity element $\mathbf{1}$.

- Next, $\bar{R}_{df}$ could grow very big and complicated as the communication pattern in the given system P gets complicated. In general cases it is hard to find generators for $\bar{R}_{df}$.

I haven't found a simple way to generate all the elements in $\bar{R}_{df}$ other than the trival but brutal way of checking all the reshufflings in $\bar{R}$. In this section, I will present some interesting elements in the group $\bar{R}_{df}$.

8.4.1 System Independent DF reshuffling

Reshuffling is defined on $\mathbf{res}(P)$ for a given DF system P, whether a reshuffling is deadlock-free or not depends on the specific system. However the dependence correlation varies from one DF reshuffling to another. We want to identify the types of DF reshufflings with the least dependence on system information.

Note: From now on, all reshuffling takes the form of operational definition given in section 3. According to the definition a reshuffling is given by giving $Q, T \in \mathbf{res}(P)$, and I refer to such a reshuffling as "a reshuffling which reshuffles Q to T".
(end of Note)

The simplest DF reshuffling on any given $\mathbf{res}(P)$ is the identity reshuffling, which changes a (DF) system $Q \in \mathbf{dfr}(P)$ to itself (in other words it is the "no-reshuffling"). It is trivial that the identity reshuffling is deadlock-free for any system.

Def: *A system independent DF reshuffling* is a deadlock-free reshuffling r which reshuffles a sequence $s = (S_1; x; S_2)$ in some process to $\hat{s} = (S_1; S_2; x)$ (where x is either a transition or a wait) such that for any imbedding process $*[T_0; s; T_3]$ and processes $p_2, \ldots, p_n$ if $*[T_0; S_1; x; S_2; T_3] \parallel p_2 \parallel \ldots \parallel p_n$ is deadlock-free then the reshuffled system $*[T_0; S_1; S_2; x; T_3] \parallel p_2 \parallel \ldots \parallel p_n$ is deadlock-free.

By the above definition, the identity reshuffling reshuffles a sequence s to itself and trivially it preserves the absence of deadlock.

Another system independent DF reshuffling is the reshuffling of consecutive transitions: reshuffling $r_{x\updownarrow}$ reshuffles the sequence $s = (S_1; x\updownarrow; y\updownarrow)$ to $\hat{s} = (S_1; y\updownarrow; x\updownarrow)$.

Def: Let P be the original directly-mapped program. Let $Q_1 = p_1 \parallel p_2 \parallel \ldots \parallel p_n$ be a program in $\mathbf{dfr}(P)$ with process p_1 as follows:

1. $p_1 = *[T_0; x\updownarrow; y\updownarrow; T_1]$, $r[Q_1][p_1][x\updownarrow]$ is a reshuffling such that:

$$\forall S \in \mathbf{dfr}(P),\ S \cdot r[Q_1][p_1][x\updownarrow] = \begin{cases} Q_2 & \text{if } S = Q_1 \\ P & \text{if } S = Q_2 \neq P \\ Q_1 & \text{if } S = P \neq Q_1 \\ S & \text{if } S \neq P,\ Q_1,\ Q_2 \end{cases}$$

   Here $Q_2 = *[T_0; y\updownarrow; x\updownarrow; T_1] \parallel p_2 \parallel \ldots \parallel p_n$.

2. $p_1 = *[T_0; y\updownarrow; x\updownarrow; T_1]$, where T_1 starts with a wait

$$\forall S \in \mathbf{dfr}(P),\ S \cdot r[Q_1][p_1][x\updownarrow] = \begin{cases} P & \text{if } S = Q_1 \\ Q_1 & \text{if } S = P \\ S & \text{if } S \neq P,\ Q_1 \end{cases}$$

Lemma 8.4.1 $r[Q_1][p_1][x\updownarrow] \in \bar{R}_{df}$.

Proof:

(a) Pick any $S \in \mathbf{dfr}(P)$, $S \cdot r[Q_1][p_1][x\updownarrow] \in \mathbf{dfr}(P)$.

proof: $S = \hat{p}_1 \parallel \ldots \parallel \hat{p}_n$. There are three cases:

case 1 : $S = P$, then $S \cdot r[Q_1][p_1][x\updownarrow] = Q_1 \in \mathbf{dfr}(P)$.

case 2a: $S = Q_1 = *[T_0; y\updownarrow; x\updownarrow; T_1] \parallel p_2 \parallel \ldots \parallel p_n$, then $S \cdot r[Q_1][p_1][x\updownarrow] = P \in \mathbf{dfr}(P)$.

case 2b: $S = Q_1 = *[T_0; x\updownarrow; y\updownarrow; T_1] \parallel p_2 \parallel \ldots \parallel p_n$, then $S \cdot r[Q_1][p_1][x\updownarrow] (= Q_2) = *[T_0; y\updownarrow; x\updownarrow; T_1] \parallel p_2 \parallel \ldots \parallel p_n$. Since $x\updownarrow$ and $y\updownarrow$ won't be transitions on communication variables in the same communication action, the only partial order that has been changed is from the original $x\updownarrow \prec y\updownarrow$ to $y\updownarrow \prec x\updownarrow$. The situation in which we can introduce a cycle is that from other processes we have $x\updownarrow \prec y\updownarrow$. But the other processes can only observe the transitions on x and y from the waits $[\pm x]$ and $[\pm y]$. We can have that $[\pm x]$ precedes some transtion t and that then t precedes $[\pm y]$. However,

we know a transition precedes a wait only if it is a transition on communication variables of the same communication. So we know $[\pm x] \prec t_y \prec [\pm y]$. But in the original process we have $t_y \prec x\updownarrow$ which is a contradiction. So $r[Q_1][p_1][x\updownarrow]$ is deadlock-free.

case 3 : $S = Q_2$, then $S \cdot r[Q_1][p_1][x\updownarrow] = Q_1 \in \mathbf{dfr}(P)$.

(b) $r[Q_1][p_1][x\updownarrow]$ is a bijection on the set $\mathbf{dfr}(P)$.

proof: By definition.

□

Remark: $r[Q_1][p_1][x\updownarrow]$ generates a cyclic group of order 2.
(end of Remark)

The aforementioned two types of reshuffling are the only two system independent reshufflings. Here is the brief argument to support this statement: For any reshuffling that reshuffles a sequence $(S_1; x; S_2)$ to $(S_1; S_2; x)$, we have the following four cases: postpone a wait until after some other wait, postpone a wait until after some transition, postpone a transition until after some other transition, postpone a transition until after some wait.

case 1: Reshuffle $[w]; (S; [v])$ to $(S; [v]); [w]$. It won't cause any deadlock only if $S = \varepsilon$ (i.e. S is an empty statement) or a sequence of waits; otherwise if there is some transition t in S then we can construct a system such that $\{w\}\ t\ \{\neg w\}$ holds and performing the above reshuffling will introduce deadlock.

case 2: Reshuffle $[w]; (S; t)$ to $(S; t); [w]$ where t is a transition. We can construct an environment such that $\{w\}\ t\ \{\neg w\}$ holds, and the reshuffling will violate the relationship and introduce deadlock.

case 3: Reshuffle $t; (S; u)$ to $(S; u); t$ where t, u are transitions. It won't cause any deadlock only if $S = \varepsilon$ or a sequence of transitions; otherwise if S has some wait $[w]$ then we can construct a system such that $\{\neg w\}$ is a precondition of t and the reshuffling will violate the precondition requirement and introduce deadlock.

case 4: Reshuffle $t; (S; [w])$ to $(S; [w]); t$. Similar argument as in case 3, we can construct a case such that $\{\neg w\}$ is a precondition of t.

Since we may always combine consecutive waits into one wait, there is no reshuffling of consecutive waits, which leaves the reshuffling of consecutive transitions as the only system independent DF reshuffling besides the identity reshuffling.

8.4.2 Environment Independent DF Reshuffling

A larger category of system-independent DF reshufflings is the environment independent DF reshuffling.

Def: *An environment independent DF reshuffling* is a deadlock-free reshuffling r which reshuffles a process $p_1 = *[T_0; S_1; x; S_2; T_3]$ to $\hat{p}_1 = *[T_0; S_1; S_2; x; T_3]$ (where x is either a transition or a wait) such that for any processes $p_2, \ldots, p_n$ if $p_1 \parallel p_2 \parallel \ldots \parallel p_n$ is deadlock-free and the reshuffling r meets the constraints on the communication variables of process p_1 then the reshuffled system $\hat{p}_1 \parallel p_2 \parallel \ldots \parallel p_n$ is deadlock-free.

The system independent DF reshuffling is a subset of the environment independent DF reshuffling which has no constraints on the communication variables. The following are some simple and useful environment independent reshufflings that belong to $\bar{R}_{df}$.

Postpone Waits

Def: Let P be the original directly-mapped program. Let $Q_1 = p_1 \parallel p_2 \parallel \ldots \parallel p_n$ be a program in $\mathbf{dfr}(P)$ with a process p_1 as follows:

1. $p_1 = *[T_0; [w]; t; \{w\}T_1]$, where t is a single transition.
   $r[Q_1][p_1][w]$ is a reshuffling such that:

$$\forall S \in \mathbf{dfr}(P),\ S \cdot r[Q_1][p_1][w] = \begin{cases} Q_2 & \text{if } S = Q_1 \\ P & \text{if } S = Q_2 \neq P \\ Q_1 & \text{if } S = P \neq Q_1 \\ S & \text{if } S \neq P,\ Q_1,\ Q_2 \end{cases}$$

   Here $Q_2 = *[T_0; t; [w]; T_1] \parallel p_2 \parallel \ldots \parallel p_n$.

2. $p_1 = *[T_0; [w]; t; \{\neg w\}T_1]$, where T_1 starts with a transition

$$\forall S \in \mathbf{dfr}(P),\ S \cdot r[Q_1][p_1][w] = \begin{cases} P & \text{if } S = Q_1 \\ Q_1 & \text{if } S = P \\ S & \text{if } S \neq P,\ Q_1 \end{cases}$$

Lemma 8.4.2 $r[Q_1][p_1][w] \in \bar{R}_{df}$.

Proof:

(a) Pick any $S \in \mathbf{dfr}(P)$, $S \cdot r[Q_1][p_1][w] \in \mathbf{dfr}(P)$.

proof: $S = \hat{p}_1 \parallel \ldots \parallel \hat{p}_n$. There are three cases:

case 1 : $S = P$, then $S \cdot r[Q_1][p_1][w] = Q_1 \in \mathbf{dfr}(P)$.

case 2a: $S = Q_1 = *[T_0; [w]; t; \{\neg w\}\ T_1] \parallel p_2 \parallel \ldots \parallel p_n$, then $S \cdot r[Q_1][p_1][w] = P \in \mathbf{dfr}(P)$.

case 2b: $S = Q_1 = *[T_0; [w]; t; \{w\}\ T_1] \parallel p_2 \parallel \ldots \parallel p_n$, then $S \cdot r[Q_1][p_1][w](= Q_2) = *[T_0; t; [w]; T_1] \parallel p_2 \parallel \ldots \parallel p_n$. We know $Q_1 \in \mathbf{dfr}(P)$ and Q_1 has $w \prec t$ as a generator of partial orders while Q_2 doesn't. So the successor relations in Q_2 are a subset of the successor relations in Q_1. Since Q_1 has no cycles in its CDG, neither does Q_2. Therefore, $Q_2 \in \mathbf{dfr}(P)$.

case 3 : $S = Q_2$, then $S \cdot r[Q_1][p_1][w] = Q_1 \in \mathbf{dfr}(P)$.

(b) $r[Q_1][p_1][w]$ is a bijection on the set $\mathbf{dfr}(P)$.

proof: By definition.

□

Remark: $r[Q_1][p_1][w]$ generates a cyclic group of order 2.
(end of Remark)

The proof works for any arbitrarily picked system Q_1 and process p_1. Therefore we have shown that postponing waits preserves the absence of deadlock.

Reshuffle Local Variables

Def: Let P be the original directly-mapped program. Let $Q_1 = p_1 \parallel p_2 \parallel \ldots \parallel p_n$ be a program in $\mathbf{dfr}(P)$ with a process $p_1 = *[T_0; U_x; T_1; D_x; t; T_2]$, where t is a transition on the local variables[2]. Then $r[Q_1][p_1][D_x]$ is a reshuffling such that:

$$\forall S \in \mathbf{dfr}(P),\ S \cdot r[Q_1][p_1][D_x] = \begin{cases} Q_2 & \text{if } S = Q_1 \\ P & \text{if } S = Q_2 \neq P \\ Q_1 & \text{if } S = P \neq Q_1 \\ S & \text{if } S \neq P,\ Q_1,\ Q_2 \end{cases}$$

[2]in contrast to communication variables or shared variables

Here $Q_2 = *[T_0; U_x; T_1; t; D_x; T_2] \parallel p_2 \parallel \ldots \parallel p_n$.

Lemma 8.4.3 $r[Q_1][p_1][D_x] \in \bar{R}_{df}$.

Proof:

(a) Pick any $S \in \mathbf{dfr}(P)$, $S \cdot r[Q_1][p_1][D_x] \in \mathbf{dfr}(P)$.

proof: $S = \hat{p}_1 \parallel \ldots \parallel \hat{p}_n$. There are three cases:

case 1: $S = P$, then $S \cdot r[Q_1][p_1][D_x] = Q_1 \in \mathbf{dfr}(P)$.

case 2: $S = Q_1 = *[T_0; U_x; T_1; D_x; t; T_2] \parallel p_2 \parallel \ldots \parallel p_n$, then $S \cdot r[Q_1][p_1][D_x](= Q_2) = *[T_0; U_x; T_1; t; D_x; T_2] \parallel p_2 \parallel \ldots \parallel p_n$. Here we change the partial order $D_x \prec t$ to $t \prec D_x$. It will cause a cycle if some other processes have $D_x \prec t$. However since t is only a transition on some local variable, its value can not be observed by the other processes. Therefore there doesn't exist the partial order $D_x \prec t$. Therefore $Q_2 \in \mathbf{dfr}(P)$.

case 3: $S = Q_2$, then $S \cdot r[Q_1][p_1][D_x] = Q_1 \in \mathbf{dfr}(P)$.

(b) $r[Q_1][p_1][D_x]$ is a bijection on the set $\mathbf{dfr}(P)$.

proof: By definition.

□

Reshuffle Communications on Internal Channels

Def: Let P be the original directly-mapped program. Let $Q_1 = p_1 \parallel p_2 \parallel \ldots \parallel p_n$ be a program in $\mathbf{dfr}(P)$ with a process $p_1 = *[T_0; U_x; T_1; D_x; t; T_2]$, X is an internal channel introduced by process decomposition. Then $r[Q_1][p_1][C_x]$ is a reshuffling such that:

$$\forall S \in \mathbf{dfr}(P),\ S \cdot r[Q_1][p_1][C_x] = \begin{cases} Q_2 & \text{if } S = Q_1 \\ P & \text{if } S = Q_2 \neq P \\ Q_1 & \text{if } S = P \neq Q_1 \\ S & \text{if } S \neq P,\ Q_1,\ Q_2 \end{cases}$$

Here $Q_2 = *[T_0; U_x; T_1; t; D_x; T_2] \parallel p_2 \parallel \ldots \parallel p_n$.

Lemma 8.4.4 $r[Q_1][p_1][C_x] \in \bar{R}_{df}$.

Proof:

(a) Pick any $S \in \mathbf{dfr}(P)$, $S \cdot r[Q_1][p_1][C_x] \in \mathbf{dfr}(P)$.

proof: $S = \hat{p}_1 \parallel \ldots \parallel \hat{p}_n$. There are three cases:

case 1: $S = P$, then $S \cdot r[Q_1][p_1][C_x] = Q_1 \in \mathbf{dfr}(P)$.

case 2: $S = Q_1 = *[T_0; U_x; T_1; D_x; t; T_2] \parallel p_2 \parallel \ldots \parallel p_n$, then $S \cdot r[Q_1][p_1][C_x](= Q_2) = *[T_0; U_x; T_1; t; D_x; T_2] \parallel p_2 \parallel \ldots \parallel p_n$. Here the partial order $D_x \prec t$ is changed to $t \prec D_x$. It will cause a cycle if some processes have the partial order $D_x \prec t$. However, since X is a communication on the internal channel introduced by process decomposition, the other process using X communication is therefore of the form $*[\bar{X} \longrightarrow \ldots; D_x]$. So D_x can always be completed and there exists no partial order $D_x \prec t$. Therefore $Q_2 \in \mathbf{dfr}(P)$.

case 3: $S = Q_2$, then $S \cdot r[Q_1][p_1][C_x] = Q_1 \in \mathbf{dfr}(P)$.

(b) $r[Q_1][p_1][C_x]$ is a bijection on the set $\mathbf{dfr}(P)$.

proof: By definition.

□

Reshuffle Communications with Disjoint Environments

Def: Given a communication X and the system configuration graph for an SLHE system P, define $E(X)$ as the *set of processes (nodes) in the connected subgraph wherein X resides.*

Lemma 8.4.5 *Given a sequence of two communications $(X; Y)$ in a process p of the SLHE system P, if $E(X) \cap E(Y) = p$, then for any reshuffling r, $P \cdot r \in \mathbf{dfr}(P)$.*

Proof: Because $E(X) \cap E(Y) = p$, communications X and Y are only ordered via process p. Therefore no ordering exists between X and Y other than that in p. Any reshuffling of the sequence $(X; Y)$ only changes the sequencing between X and Y in the process p, thus won't cause any deadlock.

□

Continuity of Reshuffling

Def: Let Q be an SLHE system $Q = p_1 \parallel \ldots \parallel p_n \in \mathbf{res}(P)$ with $p_1 = *[S_0; t; S_1; S_2; S_3]$ where t is a transition. Let r^* be a deadlock-free reshuffling such that $Q \cdot r^* = *[S_0; S_1; S_2; t; S_3] \parallel p_2 \parallel \ldots \parallel p_n$, then the reshuffling $(r|r^*)$ is a reshuffling defined as follows:

$$Q \cdot r|r^* = *[S_0; S_1; t; S_2; S_3] \parallel p_2 \parallel \ldots \parallel p_n$$

Lemma 8.4.6 *For any* $Q \in \mathbf{dfr}(P)$, *if* $Q \cdot r^* \in \mathbf{dfr}(P)$, *then* $Q \cdot r|r^* \in \mathbf{dfr}(P)$.

Proof: Suppose that $Q \cdot r|r^*$ is not deadlock-free, then its corresponding CDG must have a cycle (Chapter 6), i.e. there exists a finite set of transitions $t_1, t_2, \ldots, t_n$ such that $t_1 \prec t_2 \prec \ldots \prec t_n \prec t_1$. The cyle must contain transitions from the sequence $(S_1; t; S_2)$ since the CDG of Q is acyclic. Furthermore the cycle must contain a transition of type t (because otherwise the cycle remains after we project out transitions of type t, which implies the CDG of Q has a cycle too). Without loss of generality, suppose that $t_1 = \langle t, k \rangle$. There must exist a transition either from S_1 or S_2 in the cycle.

- Suppose there is a transition t_k from S_1 such that $t_k \prec t$ is imposed by the sequence $(S_1; t; S_2)$, this implies that $p_2 \parallel \ldots \parallel p_n$ has the partial order $t \prec t_k$. This contradicts the fact that the CDG of $Q \cdot r^*$ is acyclic.
- Suppose there is a transition t_k from S_2 such that $t \prec t_k$ is imposed by the sequence $(S_1; t; S_2)$, this implies that $p_2 \parallel \ldots \parallel p_n$ has the partial order $t_k \prec t$. This contradicts the fact that the CDG of Q is acyclic.

Therefore the CDG of $Q \cdot r|r^*$ is acyclic, which implies that the system is deadlock-free. □

Chapter 9

Conclusion

In this treatise, I have developed a theory for detecting whether or not a reshuffling introduces any deadlock. The theory can be applied to any system which is composed of processes that can be described either by straight-line handshaking expansions or by guarded commands on the HSE level. The main contribution of the work is:

- Deadlock in a computation of a closed system is characterized by cycles in the corresponding communication dependency graph. A system is deadlock-free if and only if the CDGs of all its valid computations are acyclic. Also I have proved that any closed SLHE system has a unique computation.

- CDGs for a computation are infinite graphs which makes the theory impossible to use in practice. I have developed a transformation method from a closed SLHE system to a corresponding repetitive system, such that the closed SLHE system is deadlock-free if and only if the corresponding repetitive system is deadlock-free, which reduces the cycle detection in an infinite graph to cycle detection in a finite graph.

- For a general system with SLHE processes and GC processes with deterministic choices there is a unique computation. Furthermore, the general system can be transformed to an SLHE system which shares the same computation. This result allows us to perform the liveness analysis developed for SLHE systems after transforming the general system to an SLHE system.

- I have shown that the set of deadlock-free reshufflings for a given SLHE system forms a group. Therefore, more deadlock-free reshufflings can

be generated by the composition of known deadlock-free reshufflings. I studied some interesting cases of deadlock-free reshufflings.

This research is a preliminary work to study the effect of reshuffling on system deadlock. It is important because reshuffling is a vital and widely-used transformation step in the Martin synthesis method; unfortunately corretness is not guaranteed through reshuffling, one of the major concerns being the introduction of deadlock. Algorithms have been developed to automate the deadlock detection procedure. Tools are under development to aid the reshuffling phase of the Martin synthesis method in designing VLSI systems, which will be eventually incorporated into CAST (the Caltech Asynchronous Synthesis Tool).

Future topics include:

- Generalize the detection method to systems with non-deterministic choices.
- Modeling environments with non-deterministic behavior, and studying the selection of a set of environment scenarios so that a system is deadlock-free in general if the computations with this set of environment scenarios are deadlock-free.
- Find more DF-reshufflings in the group R_{df}, so that we can generate new DF-reshufflings from the known DF-reshufflings.

Finally, I want to thank my advisor, Dr. Alain Martin, for having invented this wonderful synthesis method for asynchronous VLSI design. Many a time I can't help admiring the power of this method in the design and the formality which can't be found in other circuit designing methods. Also I want to thank Dr. Martin for giving me this interesting research topic.

Bibliography

[1] Steven Burns. *Performance Analysis and Optimization of Asynchronous Circuits.* Ph.D. Thesis, Caltech, 1991.

[2] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[3] D.S. Dummit and R.M. Foote. *Abstract Algebra.* Prentice-Hall, Englewood Cliffs, N.J. 1991.

[4] Marcel van der Goot *A Semantic Model and Formal Transformations for VLSI Synthesis.* Ph.D. Thesis, Caltech, 1995.

[5] C.A.R.Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, N.J., 1985.

[6] Tak Kwan Lee. *Performance Analysis and Optimization of Data-Dependent and Inherently Disjunctive Asynchronous Circuits.* Ph.D. Thesis, Caltech, 1995.

[7] A.J. Martin. *An Axiomatic Definition of Synchronization Primitives.* Acta Informatica 16, 219-235, 1981.

[8] A.J. Martin. *Programming in VLSI: From Communicating Processes To Delay-Insensitive Circuits.* Caltech-CS-TR-89-1.

[9] A.J. Martin. *The Limitations to Delay-Insensitivity in Asynchronous Circuits.* Caltech-CS-TR-90-02.

[10] A.J. Martin. *Asynchronous Datapaths and the Design of an Asynchronous Adder* Formal Methods in System Design, 1: 117-137, 1992.

[11] A.J. Martin. *Synthesis of Asynchronous VLSI Circuits.* Caltech-CS-TR-93-28.

[12] J. von Wright. *A Lattice-theoretical Basis for Program Refinement.* Ph.D. Thesis, Abo Akademi, 1990.

Jessie Xu

April 21, 1995

Some comments on the MIPs design:

1. The goal I understand is to design *an asynchronous version of MIPs R3000.* Therefore it is necessary to make clear our concept about MIPs R3000 architecture, for example, the same instruction set, the same exception mechanism etc. Also it is helpful to document the features that we don't consider as an architectural aspect but a design choice, such as the pipeline depth, the functionality of the $CP0$ unit etc., document the difference between this design and others so that we can defend the design when someone criticizes the design to be a MIPs R3500.

   The modifications of the basic architectural features are expected as the design goes on and with the growth of our understanding about the MIPs architecture, so good documentation is important to make our partyline clear and observed by the whole design team always.

   The documentation is better in English, or some other common language (not CSP which is understood by an exclusive group of researchers). And the partyline is better made as explicit and readable as possible so that no induction needs to be made to get to the points.

2. Documentation is very important. This is a lesson learned by students, designers thru their past design experience. A good "work-diary" (or chronic documentation) would help:

   - Record the decisions made which makes the reviewing of the reasons of decisions and changes possible, also it will benefit the future designs.
   - Record the problems about the tools etc, so that they can be fixed for future use.
   - The documentation would make it easy to write a book like: AMIPs Rx000 Architecture.

3. Another goal for the design of Asynchronous MIPs is to show the design methodology is efficient, i.e. takes fewer man-years, and that the tools are useful. In order to achieve this, we need to :

   - Keep a good diary of what each worker is working on, time spent on each design phase.

- Good interface between workers. A central coordinator (like Rajit's job) is important: each small decision would be reported to the coordinator and be well-documented and made available to the whole team.

  Another important job is to set some standards, such as the naming standards etc, so that everybody could make his design standardized.

- The support work. The design needs the support of tools etc so it is an important piece of work to prepare the tool for later use. However, this job can be done concurrently with the upper-level specification. Now the assignment of the whole design work is according to different function blocks. But there are times that you are waiting for some information to carry on your work. These times can be used to prepare yourself for the tools or help fix the tools etc.

(end of comments)